Low Cost Management of Replicated Data

Thomas A. Joseph
PH.D Thesis

TR 85-712
November 1985

TECHNICAL REPORT

Department of Computer Science
Cornell University
Ithaca, New York

12

Low Cost Management of Replicated Data

Thomas A. Joseph
PH.D Thesis

TR 85-712
November 1985

Contract MDA-903-85-C-0124
ARPA Order No. 5378

Department of Computer Science
Cornell University
Ithaca, NY 14853

LOW COST MANAGEMENT OF REPLICATED DATA

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Thomas A. Joseph

January 1986

LOW COST MANAGEMENT OF REPLICATED DATA

Thomas A. Joseph, Ph.D.
Cornell University 1986

When data are replicated in a distributed system, it is necessary to ensure that different copies of the same data are kept consistent with respect to one another. This could lead to a substantial performance degradation (latency) when operations are performed on replicated data. Even if a copy of the required data is available at the site where an operation is performed, it may be necessary to communicate with other sites where copies of the same data reside, in order to ensure that consistency is not violated. Performance suffers because message transmission times are typically much higher than local computation times.

In this thesis, we first present an object-based model of a distributed system. We then use this model to show that data can be replicated in a manner that does not incur the latency cost described above. Further, we prove that it is possible to transmit the information required to maintain consistency by piggybacking it upon synchronization messages that would be sent even if data were not replicated. The replication method hence does not require any additional messages. We describe actual implementations of this method and discuss their behavior in conjunction with roll-back and roll-forward failure handling mechanisms. Finally, we compare the performance of one such implementation with a more synchronous implementation and demonstrate that our method performs substantially better.

Biographical Sketch

Thomas A. Joseph was born on March 1, 1958. He received his bachelor's degree in Mechanical Engineering from the Indian Institute of Technology, Madras, in 1980. In the course of his undergraduate life, he succumbed to the joys of coding FORTRAN programs on punch cards, running them on a batch-processing system, waiting a day to receive the output, and re-submitting them with the statement numbers correctly punched between columns 1 and 5. The experience convinced him to give up the greasy and noisy world of machine tools for the temperature- and humidity-controlled world of computers. In the fall of 1980, he enrolled in the Ph.D. program in Computer Science at Cornell University.

To his initial disappointment (and subsequent relief), he discovered that there was more to computer science than computer programming. In due course, his interest focused on the areas of computer networks, distributed systems, fault-tolerant programming, and program verification. In 1983, he received his M.S. in Computer Science from Cornell University. Since then, he has been a member of the *ISIS* project, which aims to develop a system to aid the construction of fault-tolerant software for distributed systems. He is a member of the Association of Computing Machinery.

Acknowledgements

First of all, I wish to express my gratitude to my advisor, Ken Birman, for his constant attention and guidance. Without his encouragement and inducements, threats and ultimatums, I would not have been able to complete this thesis. He also presented me a fine bottle of port on my birthday. My thanks also go to Fred Schneider, who was responsible for helping me cross that painful bridge between taking courses and the then nebulous endeavor called "doing research." I am also grateful to him for reading a draft of this thesis and offering many helpful suggestions. Thomas Räuchle, Amr El-Abbadi and Sam Toueg receive my thanks for numerous discussions, which have had an influence on various aspects of my work. I also thank John Gilbert and H. C. Torng for being members of my committee.

There are a number of people without whose support and friendship I would never have been able to persevere until the end of my doctoral program. Somnath was a friend at a time when I was adjusting and settling down in a new environment. Pradeep, Dinesh, and Dharshan offered generous portions of their time and patience in helping me raise my spirits, both distilled and distressed. Rogério was responsible for many happy hours spent in his company. Vineeta's unconditional friendship did a lot to make my life more pleasant. Maria was – and always will be – very special to me. William J. Manos supplied endless cups of late night coffee and an atmosphere for interminable discussions (we never did figure out the meaning of life, did we, Dinesh?). (Leena-Maija, let's be honest. Without your friendship – exciting and enjoyable as it has been – I *still* would have been able to complete my Ph.D. So I'll say nothing about extensive giggling, gummi novelties, sloths, Garlickson, leipa, or anything else.) Thank you seems a small thing to say to all of you, but you know what I mean...

Finally, I wish to acknowledge the support of the National Science Foundation under grant DCR-8412582 and that of the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124. The views, opinions and findings contained in this report are those of the author and should not be construed as an official Department of Defense position, policy, or decision.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Distributed Computer Systems

Early computer systems consisted of a powerful centralized computer accessed by means of relatively unsophisticated terminals. As a result of advances in computer hardware and the development of the personal computer or work-station, it is now possible to place upon one's desk-top the processing power that used to be available only in large mainframe computers. This has led to the growth of *distributed systems*. A distributed system consists of a number of independent processing *sites*, interconnected by means of a communications network.[1] Each site possesses memory units to store data and processing units to access and manipulate data stored at that site. The network enables sites to send messages to one another and can be used to access data stored at a remote site.

A distributed system has a number of advantages over a centralized one. In a distributed system, each user can be provided with the processing power of a separate computer at relatively low cost. Users can share common services like a file system, specialized databases, or high quality output devices, which may be located at remote sites and accessed using the network. Thus, the cost of such services is amortized over all the sites. A distributed system also has performance

[1]The results in this work are equally valid if the term *site* is replaced by *process at a site*, provided that processes communicate by sending messages to one another and share no memory.

advantages. In a centralized system, all operations are carried out at the same site. Hence, the central site must be flexible enough to respond to different types of requests and cannot be optimized for a single type of operation. In a distributed system, on the other hand, sites can have different capabilities and system performance can be improved by executing operations at sites best suited to do so. Further, if the work-load is uneven, bottlenecks can form in a centralized system during times of peak load, with the system remaining relatively idle at other times. In a distributed system, the work-load can be spread out more evenly by assigning jobs to lightly loaded sites. Another advantage of distributed systems is that they are easy to expand: more sites can be added as required. A distributed system is also potentially more robust than a centralized one because if a site fails, the other sites can continue to function. In addition, operational sites can assume some of the work that would have been performed at a failed site. For this to be possible, information stored at the failed site must be available to an operational one. This motivates the need for *replicated data*.

1.2. Replicated data

A distributed system permits copies of the same information to be stored at more than one site. If a data item is accessed frequently from a set of sites, a copy can be stored at each of these sites, and each site could access its local copy. Since the time required for a message to be sent between sites is typically much higher than that required for local processing, accessing a local copy of data in a distributed system is much faster than accessing data stored at a remote site. Replicated data can also be used to achieve fault-tolerance. If a site fails, data stored at that site may become unavailable to the rest of the system. However, if data are replicated, the system can continue operating by accessing another copy of the data instead.

Replicated data, however, is not without its costs. The obvious ones are the costs of storing more than one copy of the same data and the processing costs involved with updating several copies instead of one. These costs are unavoidable. Other costs arise from the need to keep different copies of replicated data consistent. When one copy is changed, the other copies must be updated to reflect this change. Otherwise, actions taken by a site based on one copy of replicated data may not in agreement with actions taken by a different site based on another copy. A simple example will illustrate this problem.

Consider a replicated data object that represents airline seat reservations. If one copy of the object is updated whenever a reservation is made, but the change is not propagated to the other copies, travel agents using different copies may make more reservations than there are seats available, or assign the same seat to different passengers. This outcome could be embarrassing to the airline and inconvenient for the passengers. In this example, the copies of the replicated data object are *inconsistent*.

The cost of keeping copies of replicated data consistent manifests itself in two ways. First, the time taken to update replicated data is greater than that for non-replicated data. We call this effect *latency*. Compare a non-replicated implementation of the airline seat reservation object with a replicated one. In the former case, the time taken for an update by the site where the object is stored is simply the time required for local processing. On the other hand, if a user at a remote site wishes to perform an update, he or she must wait for a message transmission to take place between the sites. In the replicated case, an update made by any site must be propagated to other sites. Moreover, an update *u* cannot be performed until all the sites agree that it is safe to do so; that is, there are no ongoing updates at remote sites that must be completed everywhere before *u*. Thus, depending on the implementation, the time required to perform an update in the replicated case could be as long as the time required to perform a remote update in the non-replicated case. As a result, the response time when updating replicated data will be greater than that for updating non-replicated data, even when a copy is available locally. This latency could be avoided by decoupling remote updates from local ones, but this must be done in a way that maintains consistency.

The problem of decoupling remote updates from local ones has been addressed in the context of replicated databases, where only read and write operations are possible on the data. Traiger *et al.* show that when two-phase locking is used, remote writes can be deferred until commit time without affecting the consistency of data [45]. Eager and Sevcik describe a concurrency control method in which transactions are executed locally, with write operations propagated to remote sites later [17]. In [28], we show how to decouple remote updates in a database system that uses any concurrency control algorithm following a read-one-copy, write-all-copies rule. In this work, we generalize that result even further – beyond database systems – to systems with arbitrary types of data and more complex operations than reads and writes.

The other cost that arises from the need to maintain consistency is an increase in message traffic. This follows from the fact that information about updates must be transmitted between sites where copies of replicated data are situated. In many distributed systems, the critical cost factor is the number of messages sent, and not their size. This is especially prevalent when each message is processed by a large number of software layers before being physically transmitted or after being received. In such systems, each message sent has a relatively high fixed cost and a smaller variable cost that depends on its size. Therefore, in this thesis, we measure the cost of communication by the number of messages transmitted, rather than the total amount of information sent. This cost measure can be justified on the basis of studies such as [11].

1.3. Overview of the thesis

The aim of this ~~work~~ is to develop a method for managing replicated data that eliminates latency. Thus, our work can be viewed as a generalization of the results in [17, 28, 45] to systems with arbitrary types of data, with operations more powerful than reads and writes, and with other kinds of synchronization mechanisms than two-phase locking. Further, we show that it is possible to transmit the information required to maintain consistency using messages that would be sent between sites even if data were not replicated. In other words, the replication method requires no additional messages. The implication is that of the costs of replication described above, the only costs that need be incurred are the unavoidable costs of additional storage and local processing.

The thesis is organized as follows. In the next chapter, we present a model for distributed systems. The model was motivated by the use of logs to model database systems, extended to abstract data types as in [46], and treats replicated data in a manner similar to [25]. Chapter 3 focuses on one component of the model – the scheduler. The concept of schedulers for database systems is generalized to cover systems with arbitrary types of data and operations. The notions of conflicting operations and classes of serializable schedules are accordingly generalized. In Chapter 4, the implementation another component of the model – distributed objects – is discussed. Chapter 5 uses a result from Chapter 4 to develop two methods for managing replicated data that realize the goals described above. Since one of the reasons for replicating data is to tolerate failures, failure handling mechanisms are discussed in Chapter 7. We implemented one of the replication methods described in Chapter 5 and measured its performance. The results of these tests are presented in Chapter 7. The last chapter summarizes the work in this thesis and indicates future directions.

CHAPTER 2

Model

2.1. Introduction

The major portion of this chapter is devoted to developing a model for distributed systems. We are primarily interested in distributed systems consisting of a cluster of computers or work-stations, connected to one another by a high bandwidth local-area network like an Ethernet [36]. We assume that the sites are functionally equivalent; that is, they are capable of performing the same types of operations, though some sites may be more efficient at some operations than others. We further assume that the system is *asynchronous* – the relative speeds of computations at different sites and the times taken for message transmissions are unpredictable. Asynchronicity also implies that there is no global clock or shared memory with which different sites can coordinate their actions.

One aspect of a distributed system that we model is the synchronization between events at different sites. Typically, several sites in a distributed system cooperate to solve the same problem. This requires that events at different sites be coordinated and obey certain ordering constraints. The problem of synchronizing concurrent events is not new; it has been studied extensively in the context of operating systems [2, 16, 26]. However, differences arise when the system is distributed and asynchronous. Synchronization based on a common clock is not possible. Using shared memory locations for synchronization is impractical because of the long delays associated with sending messages to access memory at a remote site. Furthermore, methods based on message passing must be modified to take into account the fact that messages between sites take much longer than messages between processes at the same site. An important observation is that the absence of a global clock means that *any* synchronization method in an asynchronous distributed system must be based on messages sent between sites. We return to this in Section 2.6.

2.2. Overview of the system model

Our model of a distributed system is motivated by the use of logs to model database systems as in [5, 12, 18, 37, 38, 43]. In these models, all data items are of the same type – they have a single value that can be accessed using a **read** operation or overwritten using a **write** operation. We extend the idea of logs to abstract data types with arbitrary operations as in [46]. The extension to replicated data is done in a manner similar to [25]. We make use of terms like *transaction* and *serializability* from database theory, but these terms are used here in a more general context than is standard.

Our model consists of three components: distributed objects, transactions, and the scheduler. *Distributed objects* model the data-storage and manipulation aspects of a distributed system. A distributed object encapsulates some data and provides a set of *operations* to access and alter these data. The act of causing an object to perform an operation on its data is called an *invocation* of the object.[1] When an object is invoked, it provides a *result* for the invocation. The result depends on the

[1]We use the terms *object* and *distributed object* interchangeably.

operation, the value of the arguments for the invocation, and the current values of the object's data. An invocation may also cause the values of the object's data to be changed.

Each object is an instance of an *object type*. Objects of the same type each have their own copies of data, but provide the same set of operations. Finally, associated with each object O is a set of sites $Accessible_O$ from which O is *accessible*. O can be invoked only from sites in $Accessible_O$.

As an example, consider an object of type **integer**. Objects of this type encapsulate a single data item: an integer value. The operations they provide are **read**, which can be used to obtain the current value of the integer, and **write**. Invoking **write**(i) changes the value of the integer to i and returns **ok**() when the operation is completed. Thus, if the integer object *TemparatureInIthaca* is invoked to perform **write**(-20), the value of *TemperatureInIthaca* would be changed to -20 and the result would be **ok**(). We denote this as [*TemperatureInIthaca*.**write**(-20); **ok**()]. If this is followed by an invocation of the **read** operation on *TemperatureInIthaca*, the result would be **ok**(-20).

More complex objects are possible. An example is an object of type **queue**, representing an ordered list of records. Objects of this type provide the operations **insert**, **first**, and **ListQueue**. Invoking **insert**(r) inserts record r at the end of the queue, obtaining the result **ok**() when done. The invocation **first**() returns **ok**(r) if the queue is not empty and r is the first record in the queue. If the queue is empty, the result returned is **empty**(). The result of **ListQueue**() is a list of all the records in the queue, in order.

Another object type is **indexed_file**, with operations **add**, **remove**, **lookup**, and **ListFile**. An object of this type represents a file of indexed records. Invoking **add**(i, r) adds record r to the file with index i, and **remove**(i) removes the record with index i. The invocation **lookup**(i) returns the record associated with index i, if it exists in the file. If no such record exists, the result returned is **not_found**(). **ListFile**() returns a list of the records currently in the file, in the order they were added to the file.

Objects in a distributed system are likely to be accessed by a number of users concurrently. Yet, a user may wish to perform a series of operations on one or more objects without their executions being interrupted by another user. An example of where this might be necessary is given below.

Consider two objects *Account*1 and *Account*2 of type **integer**, each containing the current balance in a bank account. Assume that they initially contain $1000 and $2000 respectively, and that user A wishes to transfer $100 from *Account1* to *Account2*, while user B wishes to transfer $50 from *Account2* to *Account1*. Let their invocations be interleaved as shown in Figure 2.1. We see that although each user individually performed a correct sequence of invocations, the way in which their invocations were interleaved resulted in $2100 being placed in *Account*2 and $1050 in *Account*1. This is clearly an incorrect outcome (especially from the point of view of the bank). The system has been made inconsistent.

The system would not be inconsistent if all the invocations of user A had preceded those of user B or *vice versa*. In general, a distributed system must contain some synchronization mechanism using which a user can specify that a series of

User A	User B
[*Account*1.**read**(); **ok**(\$1000)]	
	[*Account*2.**read**(); **ok**(\$2000)]
[*Account*2.**read**(); **ok**(\$2000)]	
	[*Account*1.**read**(); **ok**(\$1000)]
[*Account*1.**write**(\$900); **ok**()]	
	[*Account*2.**write**(\$1950); **ok**()]
[*Account*2.**write**(\$2100); **ok**()]	
	[*Account*1.**write**(\$1050); **ok**()]

Figure 2.1. Bank account example

operations be executed without being interrupted by the execution of other operations. A user does this by means of *transactions*.

A transaction is simply a sequence of invocations that a user wishes to have executed as a unit. When a transaction is executed in isolation on a system in a consistent state, it is assumed to leave the system in a consistent state. Thus, in the preceding example, the sequence of invocations performed by user A is a transaction, as is the sequence performed by user B. The system may interleave the execution of invocations from different transactions, but does so only if the effect would be the same as if this interleaving did not occur. As a result, the execution of any number of transactions on a system in a consistent state leaves it in a consistent state.

The part of a distributed system that orders invocations with respect to one another is modeled by the *scheduler*. Invocations resulting from ongoing transactions are presented to the scheduler, which may accept them immediately or may chose to defer their execution. The scheduler decides on an order in which to perform invocations it accepts, and instructs objects to execute them in this order. Objects return the results of invocations directly to the user. The interaction between the components of the model is shown in Figure 2.2.

2.3. Distributed Objects

We now discuss how the behavior of a distributed object may be specified. Let $DATA_O$ be the data encapsulated by object O and OP_O be the set of operations on $DATA_O$. Let **op** denote an element of OP_O. An invocation of O at site s to perform **op**, with the appropriate number of arguments of the correct type, is

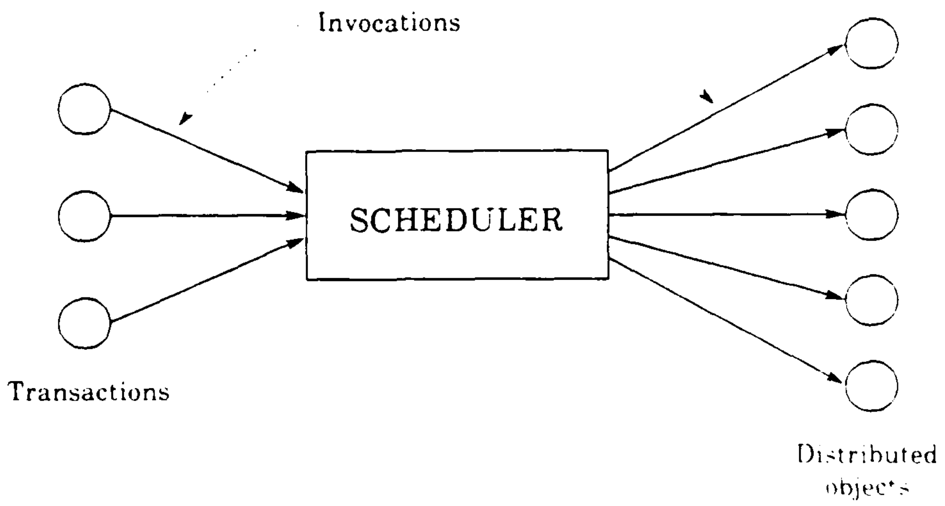

Figure 2.2. Components of the model

represented as [O.**op**(); s]. When an object executes an operation and returns a result, it is said to perform an *action*. An action is denoted [O.**op**(); **res**()], where **res**() is the result returned. A *serial computation* is a sequence of actions taken by an object, representing a series of invocations and their corresponding results. The behavior of an object is defined by its *specification* SP_O, which is the set of all correct serial computations.

Let us illustrate these definitions by an example. Consider an object x of the type **integer** discussed earlier. [x.**write**(21); **ok**()] is a possible action, denoting an invocation to change the value of x to 21 and the result **ok**() returned by x. [x.**write**(21); **ok**()] [x.**write**(15); **ok**()] [x.**read**(); **ok**(15)] is a possible serial computation denoting three successive invocations and their corresponding results. Since this serial computation reflects correct behavior for an objet of type **integer**, it would belong to the specification SP_x. On the other hand, the serial computation [x.**write**(21); **ok**()] [x.**read**(); **ok**(15)] probably would not belong to the specification, as it is not correct for the value 15 to be the result of a read immediately after the value 21 is written.

In this work, we are not concerned with the actual representation of object specifications. This may be done using logical predicates as in [27, 32], by state machines as in [46], by algebraic specifications as in [22], or by any other means. For the examples we present, we rely on the reader's intuitive understanding of the semantics of the operations discussed to decide whether a serial computation belongs to a specification or not.

A specification SP_O is said to be *complete* if for every allowable serial computation and for every operation **op** $\in OP_O$, there is another serial computation that extends the first one by an action $[O.\mathbf{op}(); \mathbf{res}()]$, for any correct set of arguments for **op**. In other words, after any sequence of invocations, a result is defined for the next invocation, whatever the operation invoked may be. Formally, a specification is said to be complete if the empty sequence $\varphi \in SP_O$ and if for all $x \in SP_O$ and all **op** $\in OP_O$, there exists a $y \in SP_O$ such that $y = x \bullet [O.\mathbf{op}(); \mathbf{res}()]$, where $\bullet$ is the concatenation operator for sequences. We assume that all specifications are complete. Since it is valid for an object to return the result **error**(), this is not a restriction.

A specification is *prefix closed* if for every $x \in SP_O$, every prefix of x also belongs to SP_O. This excludes serial computations that cannot be performed one invocation at a time, with each step resulting in a correct computation. We assume that all specifications are prefix closed.

A *deterministic object* is one that always returns the same result for each invocation when given the same sequence of invocations starting from the initial state. Given a sequence of invocations, the specification of a deterministic object contains only one serial computation in which the invocations occur in that order. If the specification contains more than one such serial computation, the object is said to be *non-deterministic*. When a non-deterministic object is given a sequence of invocations, it may return results according to any one of the applicable serial computations. Our results hold for both deterministic and non-deterministic objects.

Although the behavior of an object is specified in terms of serial computations, it is not necessary that actual invocations of the object occur serially. Even if invocations at the same site are ordered sequentially, the fact that the system is asynchronous means that invocations from different sites are unordered relative to one another, unless the scheduler explicitly orders one before the other. Thus, invocations on an object are, in general, only partially ordered. Under these conditions, we say that a distributed object behaves according to its specification if it returns results according to *some* serial computation in which the order of invocations is consistent with the partial order on actual invocations. The particular serial computation chosen would depend on how the object is implemented.

Let *PrintQueue* be an object of type **queue**, with the following invocations: [*PrintQueue*.**insert**(*WarAndPeace*); *s*], [*PrintQueue*.**insert**(*RomeoAndJuliet*); *t*], [*PrintQueue*.**ListQueue**(); *s*], and [*PrintQueue*.**ListQueue**(); *t*]. Let the scheduler order the invocations as in Figure 2.3. Each of the **ListQueue** invocations is ordered after both the **insert** invocations, but the **insert** invocations are not ordered relative to each other. Both the following serial computations would then be consistent with the partial order on invocations:

(1) [*PrintQueue*.**insert**(*WarAndPeace*); **ok**()]

[*PrintQueue*.**insert**(*RomeoAndJuliet*); **ok**()]

[*PrintQueue*.**ListQueue**(); **ok**(*WarAndPeace*, *RomeoAndJuliet*)]

[*PrintQueue*.**ListQueue**(); **ok**(*WarAndPeace*, *RomeoAndJuliet*)]

$$\begin{tikzcd}
{[\mathit{PrintQueue}.\mathbf{insert}(\mathit{WarAndPeace});\ s]} \arrow[dd] \arrow[ddr] & \\
 & {[\mathit{PrintQueue}.\mathbf{insert}(\mathit{RomeoAndJuliet});\ t]} \arrow[dl] \arrow[d] \\
{[\mathit{PrintQueue}.\mathbf{ListQueue}(\ );\ s]} & {[\mathit{PrintQueue}.\mathbf{ListQueue}(\ );\ t]}
\end{tikzcd}$$

Figure 2.3. A partial order on invocations

(2) [*PrintQueue*.**insert**(*RomeoAndJuliet*); **ok**()]

[*PrintQueue*.**insert**(*WarAndPeace*); **ok**()]

[*PrintQueue*.**ListQueue**(); **ok**(*RomeoAndJuliet*, *WarAndPeace*)]

[*PrintQueue*.**ListQueue**(); **ok**(*RomeoAndJuliet*, *WarAndPeace*)]

Hence, either result for the **ListQueue** invocations would be correct. The actual result would depend on how the object is implemented and on factors like message delays, etc. On the other hand, if the scheduler orders the **insert** invocation at t after the **insert** at s (Figure 2.4), then only (2) would be consistent with the partial order, and only one result would be possible.

The (partial) order in which invocations are performed on a distributed object O is modeled by its *history* $H_O = [I_O, \rightarrow_O]$, where I_O is a set of invocations and $\rightarrow_O$ is a partial order on them. (We use the notation $i \rightarrow_O j$ to mean $(i, j) \in \rightarrow_O$.) As described above, O provides results for its invocations based on some sequential

$$\begin{tikzcd}
{[\mathit{PrintQueue}.\mathbf{insert}(\mathit{WarAndPeace}); s]} \arrow[dd] \arrow[ddr] \arrow[dr] & \\
& {[\mathit{PrintQueue}.\mathbf{insert}(\mathit{RomeoAndJuliet}); t]} \arrow[dl] \arrow[d] \\
{[\mathit{PrintQueue}.\mathbf{ListQueue}(\,); s]} & {[\mathit{PrintQueue}.\mathbf{ListQueue}(\,); t]}
\end{tikzcd}$$

Figure 2.4. Another partial order on invocations

computation in which the invocations occur in an order consistent with the partial order $\rightarrow_O$. This is modeled by the *development*. The development D_O of an object O is a total order on the invocations in I_O; that is, it is a sequence containing all the invocations in I_O (and no others). We say that D_O is *consistent with* H_O if for all pairs of invocations i, j such that $i \rightarrow_O j$, it is true that i occurs before j in D_O. The set of actions formed by pairing each invocation in D_O with the corresponding result returned by O is called the *response* of O and is denoted R_O. Note that we do not explicitly model the state of an object. This is not necessary, since the state can be deduced from the values of H_O, D_O and R_O.

We now formalize what it means for an object to behave according to its (possibly non-deterministic) specification, when the invocations may be only partially ordered. First, its development D_O must be consistent with its history H_O. A second condition, described below, states that the response R_O agrees with the specification SP_O. Let $LegalComps_O(D_O)$ be the set of serial computations in SP_O in which the invocations occur in the same order as in D_O. (If O is deterministic, there is only one such computation and $|LegalComps_O(D_O)| = 1$.) For each computation c in $LegalComps_O(D_O)$, let $actions(c)$ be the set containing all the actions in c; that is, the order on the invocations is overlooked. Let $LegalResponses_O(D_O)$ be the set $\{actions(c) \mid c \in LegalComps_O(D_O)\}$. In other words, $LegalResponses_O(D_O)$ is the set of responses allowed by the specification, if the development is D_O. (Again, if O is deterministic, $|LegalResponses_O(D_O)| = 1$.) For O to behave according to its specification, R_O must belong to $LegalResponses_O(D_O)$.

We now present a definition that will be used later. Let $last(H_O)$ be the set of invocations in H_O that are not ordered before any other invocation. In other words, $last(H_O)$ is $\{i \mid i \in I_O \text{ and } \nexists\, j\colon i \rightarrow_O j\}$. We also extend the definition of $LegalResponses_O(D_O)$ to cover the case of a single invocation i in D_O. Let $LegalResponses_O(i,\, D_O)$ stand for the set of actions in $LegalResponses_O(D_O)$ that correspond to the invocation i. Formally, $LegalResponses_O(i,\, D_O) = \{a \mid a$ is the action corresponding to i in R for some $R \in LegalResponses_O(D_O)\}$.

2.4. Transactions

A transaction T is modeled by a set I_T of invocations and a partial order $\rightarrow_T$ on these invocations. The partial order reflects the data flow relationships between invocations, and this order must be observed in any execution of the transactions. We denote the set of all possible transactions by $TRANS$.

2.5. The scheduler

The function of the scheduler is to enforce an order between invocations from different transactions in such a way that the transactions appear to have executed independently from one another. We assume that the scheduler assigns each transaction a unique identifier called a transaction-ID. The behavior of the scheduler is modeled by a *system history* $H_{SYS} = [E_{SYS}, \rightarrow_{SYS}]$, where E_{SYS} is a set of *events* and $\rightarrow_{SYS}$ is a partial order on these events. An event has the form $[Tid, O.\mathbf{op}(), s]$ and represents the invocation $[O.\mathbf{op}(); s]$ issued by a transaction with identifier Tid. The order $\rightarrow_{SYS}$ reflects the ordering decisions made by the scheduler. For events resulting from the same transaction T, the order $\rightarrow_{SYS}$ includes all the elements in $\rightarrow_T$. In addition, $\rightarrow_{SYS}$ may contain elements relating events from different transactions.

After the scheduler has ordered invocations, it passes them on to the relevant objects for execution. The system history gives the ordering decisions made by the scheduler, while an object history gives the order in which an object receives invocations. It follows, then, that object histories can be deduced from the system history. The history of an object O consists of all invocations in H_{SYS} corresponding to O, ordered in the same way as in $\rightarrow_{SYS}$. In other words, H_O is the *projection* of H_{SYS} on O.

Finally, observe that if a global real time clock were available in the system, the events in the system history could be totally ordered according to it. It follows that each event e in E_{SYS} can be assigned a unique label $time(e)$ that can be used to place the events in this total order. This labeling may not be known to the scheduler or to any other component of the system; we merely use the fact that this labeling must exist. Given an event e, the set $before(e)$ is defined as the set $\{e' \mid time(e') < time(e)\}$.

2.6. Order in an asynchronous system

We have said that the scheduler "orders" events. In this section, we discuss what it means for the scheduler to order one event after another. Recall that in an asynchronous system there is no global clock and any synchronization between events at different sites must be based on messages sent between them. While two events at the same site can be ordered relative to each other in the normal way, the only way for a scheduler to ensure that an event e_2 at a site t is ordered after an event e_1 at a site s is to cause a message to be sent from site s to site t (perhaps indirectly via other sites) carrying the information that e_1 has occurred and e_2 may proceed. This problem is discussed in detail in [31]. We call the messages used by the scheduler to order events relative to one another *synchronization messages*.

We now formalize this notion. Let $sender(m)$ refer to the site from which a synchronization message m is sent and let $receiver(m)$ be its destination. A *message path* exists from event e_1 at site s to an event e_2 at site t either if $s = t$ and e_2 occurs after e_1 at this site, or if there is a sequence of synchronization messages $m_1, m_2, \ldots, m_n$ for which the following conditions hold.

1. $sender(m_1) = s$ and m_1 is sent from s after e_1 occurs there.

2. $sender(m_i) = receiver(m_{i-1})$ for $1 < i \leq n$, and m_i is sent after m_{i-1} is received.

3. $receiver(m_n) = t$ and e_2 occurs after m_n is received at t.

It follows from the previous discussion that two events in an asynchronous system can be ordered relative to each other only if there is a message path between them. In particular, if e_1 and e_2 belong to E_{SYS} and $e_1 \rightarrow_{SYS} e_2$, then there is a message path from e_1 to e_2. We use this fact in Chapter 5.

CHAPTER 3

The scheduler

3.1. Introduction

The scheduler orders invocations in such a way that the effects of the resulting execution are indistinguishable from one in which the transactions are executed one after another in some serial order. A history resulting from ordering invocations in this way is said to be *serializable*, and is called an *S-history*. A scheduler that produces only *S*-histories is called an *S-scheduler*. Serializability in database systems has been studied in [3, 4, 5, 12, 18, 37, 38, 43], among others. In [25, 46], the notion of serializability is generalized to abstract data types.

An *S*-scheduler could operate by fixing an order on transactions and ordering every invocation of a transaction after all invocations of transactions ordered before it. The resulting histories would be serializable, but the disadvantage of such a scheduler is clear. Only one transaction can be operational in the system at any time – every transaction must wait until all the invocations of the previous one have completed before it can proceed. This is often an unnecessary restriction. Consider, for example, a transaction that invokes an integer object *TemperatureIn-Ithaca* to perform **write**(23) and a second transaction that preforms **write**(82) on another object *TemperatureInPaloAlto*. There is no need for either transaction to wait for the other to complete, since they operate on different objects and the results of their invocations would be the same even if they are executed concurrently. Because the scheduler described above places an order between such invocations, it could cause unnecessary delays in processing transactions. It also takes no advantage of the replicated processing power available in a distributed system. In general, it is desirable for a scheduler to permit as many invocations as possible to execute concurrently, while still maintaining serializability.

The level of concurrency permitted by a scheduler can be measured by studying the class of histories it generates. This problem has been studied in the context of databases, where the operations are **read** and **write** on single valued data items. Kung and Papadimitriou measure the performance of a scheduler by its *fixpoint set*, which corresponds to the set of histories allowed by the scheduler [29]. The larger this set, the more concurrency the scheduler allows. The properties of different sub-classes of serializable histories (e.g. *DSR*, *VSR* and *CPSR*) have been studied in [23, 37, 38, 48]. In this chapter, we generalize these ideas to the case of arbitrary data types, with general operations on the data. We define the notion of *conflicting invocations* in this setting, and introduce a class called *C*-histories, which is a generalization of the classes *DSR* or *CPSR* in database theory.

3.2. When can invocations be left unordered?

Invocations have to be ordered relative to each other when their effects, as perceived by a user, could depend on the order on which they are executed. When a user cannot distinguish the relative order of a pair of invocations based on their results or the results of future invocations, the invocations can be executed in parallel. It is not incorrect for a scheduler to order such invocations, but this lowers the level of concurrency in the system.

A simple case where it is unnecessary to order invocations is when they access different objects. The result of one invocation is independent of the other and hence of the order in which they are carried out. Moreover, the data of each object are left in the same state regardless of the order in which the invocations are performed. Consequently, the result of no future invocation on either object will depend on their order. The order is hence indistinguishable to a user.

Sometimes it is not necessary to order operations even when they access the same object. Consider two transactions, each performing a **read** operation on the object *LottoWinner* of type **integer**. The result returned for each invocation is the same, regardless of the order in which the reads occur. Moreover, *LottoWinner* remains in the same state in either case, so the result of no subsequent invocation on *LottoWinner* could depend on this order. This is an example where invocations need not be ordered relative to each other because of the semantics of the operations in question (in this case, the semantics of **read**).

Consider an object x of type **number**, which provides an operation **multiply**. Invoking **multiply**(y) sets the value of x to the value obtained by multiplying the current value of x by y. The result returned is the new value of x. Now, if one transaction performs x.**multiply**(2) and another one performs x.**multiply**(3), the results would, in general, differ depending on the order in which these invocations are carried out. However, if the current value of x was 0, the order in which the invocations took place would be indistinguishable to a user. If the scheduler knew the value of x, it could leave the invocations unordered. Thus, knowledge of the current state of an object can be used to avoid ordering operations.

Again, if two transactions were each performing a **write** on an object of type **integer**, their invocations would normally have to be ordered relative to each other, because the result of a subsequent **read** operation could depend on the order. However, if it were known that the values being written were the same, the order of invocations would be immaterial. In this case, the arguments for the invocation determine whether they can be left unordered.

Finally, consider two invocations if the **add** operation on an object of type **indexed_file**, described earlier. If it is known that no transaction performs a **List-File** operation, then the invocations need not be ordered, as this is the only operation whose result depends on the way in which **add** operations are ordered relative to one another. Here, knowledge about the future behavior of transactions can be used to avoid ordering invocations.

We see from these examples that knowledge of the semantics of operations, the current state of objects, the arguments for a particular invocation, or the future behavior of transactions can all be used by the scheduler to increase concurrency by not ordering certain operations relative to others. A scheduler that maintains and uses all this information would, unfortunately, be extremely complex and inefficient. A practical scheduler uses some, but not all, of this information and hence may order more operations than strictly necessary for serializability. Examples of schedulers for database systems are given in [1, 12, 19, 29, 37, 42, 46]. Kung and Papadimitriou [29] model the level of knowledge available to a database scheduler and present optimal schedulers for different levels of knowledge.

Recall that the aim of this work is to develop an efficient method of managing replicated data. In particular, we present an implementation in which the information transmitted to keep data consistent is included in synchronization messages that must be sent by the scheduler even if data are not replicated. To be completely general, we allow for the case where the scheduler may send as few messages as possible. We allow for the possibility that the scheduler may have a high level of knowledge, which it uses to limit the number of invocations it orders. In Section 2.6, we observed that a scheduler in an asynchronous system can order invocations only by sending messages between sites. Hence, if the scheduler orders fewer operations, it will send fewer synchronization messages. We show that, even under these conditions, the messages that the scheduler must send are sufficient for the implementation of replicated data to operate correctly. A practical scheduler, operating with less knowledge, can only send more synchronization messages. As a result, the implementation remains valid.

In terms of our model, we assume that the scheduler could have knowledge of all object specifications (and hence of the semantics of all operations), the current system history (which together with the object specifications yields information about the current state of each object), the arguments for each invocation, and the set *TRANS* of all possible transactions (which amounts to information of possible future behavior of transactions). On the other hand, the scheduler can have no knowledge of the actual implementation of an object. The scheduler bases its decisions on the fact that objects meet their specifications, but where a specification can be satisfied in more than one way, the scheduler can make no assumptions about which way an object choses. In terms of the model, while the scheduler may have knowledge of the history H_O of an object O (this is simply the projection of H_{SYS} on O), all it knows about the development D_O is that it is consistent with H_O. Further, it has no knowledge of the response R_O except that it belongs to $LegalResponses_O(D_O)$ for some D_O consistent with H_O. However, if H_O and the specification SP_O are such that only one value of D_O or R_O satisfy the conditions above, the scheduler may make use of this knowledge.

We now characterize the invocations that a scheduler must order, given that it has the kind of knowledge discussed above. Any practical scheduler, having less knowledge, orders a superset of these invocations. First, we formalize the notion of serializability.

3.3. Serializability

Let H_{SYS} be a system history and let H_O be the projection of H_{SYS} on object O. Let D_O be any development consistent with H_O (not necessarily the actual development of O) and let R_O be any response belonging to $LegalResponses_O(D_O)$. Let $\rightarrow_{SER}$ be a total order on the transaction-ID's in H_{SYS}. $\rightarrow_{SER}$ is called a *serialization order* for H_{SYS}, and represents the order in which a user perceives transactions to have executed (although their invocations may actually be interleaved in H_{SYS}). H_{SYS} is serializable if the following condition holds.

Define $D = Reorder(D_O, \rightarrow_{SER})$ as the development formed by placing the invocations in D_O in the order that would have resulted had the transactions actually executed in the order specified by $\rightarrow_{SER}$. That is, if $i_1 = [O.\mathbf{op}_1(), s]$ in D_O corresponds to $[Tid_1, O.\mathbf{op}_1(), s]$ in H_{SYS} and $i_2 = [O.\mathbf{op}_2(), t]$ in D_O corresponds to event $[Tid_2, O.\mathbf{op}_2(), t]$ in H_{SYS}, and if $Tid_1 \rightarrow_{SER} Tid_2$, then ι_1 occurs before ι_2 in $Reorder(D_O, \rightarrow_{SER})$. H_{SYS} is serializable if the response R_O is legal based on the development $Reorder(D_O, \rightarrow_{SER})$. That is, H_{SYS} is serializable if the response based on D_O is indistinguishable from one based on a development that would have resulted had the transactions executed sequentially.

Formally, H_{SYS} is serializable if $\exists \rightarrow_{SER}$: [$\forall$ objects O and $\forall$ D_O consistent with the projection of H_{SYS} on O: $LegalResponses_O(Reorder(D_O, \rightarrow_{SER})) \subseteq LegalResponses_O(D_O)$].

3.4. Sub-classes of serializable histories

In [37], it is shown that the problem of recognizing whether a given database history is serializable is NP-complete. It is also shown that a scheduler that generates histories that span the complete set of S-histories must take time exponential in the number of invocations (unless $P = NP$). An *efficient scheduler* (one that takes time polynomial in the number of invocations) generates only a subset of all possible S-histories. This sacrifices some of the concurrency available in the system because certain executions that are actually serializable are disallowed.

Different sub-classes of S-histories and their corresponding schedulers have been studied in the context of databases [23, 37]. Papadimitriou [37] shows that the class DSR encompasses the classes of histories generated by most known scheduling disciplines like two-phase locking, timestamping, etc. The class DSR is based on the notion of *conflicting invocations*. Two invocations in a database system conflict if they both access the same object and one of them performs a **write** operation. In a history belonging to the class DSR any two conflicting invocations must be ordered relative to each other. We generalize this idea to distributed objects, and call the corresponding class of histories *C-histories*. A scheduler that generates C-histories is called a *C-scheduler*. We assume that the scheduler in our system is a C-scheduler.

3.5. C-histories

An invocation is said to conflict with another if the order in which the invocations are executed could make a difference to their results or to the result of some future invocation. The earlier discussion on serializability showed that whether an invocation conflicts with another could depend on the semantics of the operations in question, the current state of the objects, the arguments for the invocations and the future behavior of transactions. We take this into account in the definition of conflict given below. In a C-history, every invocation is ordered relative to any invocation that it conflicts with. This does not preclude other orderings, but merely stipulates a set of orderings that must be present in a C-history.

We now define conflict formally. Define the *extension* of a system history to be any history that could result from the completion of ongoing transactions and/or the execution of any new transactions from $TRANS$. Let $e_2 = [Tid_2, O.\mathbf{op}(), s]$ be an event in H_{SYS} and let $e_1 = [Tid_1, O.\mathbf{op}(), t]$ be any other event in $before(e_2)$ that has a different transaction-ID, but invokes the same object. Let H'_{SYS} be an extension of $before(e_2)$ that contains e_2 and in which e_2 is not ordered relative to e_1. Let H'_O be the projection of H'_{SYS} on O. Let D'_O be any development consistent with H'_O of the form $\gamma \iota_1 \iota_2 \delta$, where ι_1 and ι_2 are the invocations corresponding to e_1 and e_2 respectively, and γ and δ are sequences of invocations. Then e_2 conflicts with e_1 if $LegalResponses_O(\gamma i_1 i_2 \delta) \neq LegalResponses_O(\gamma i_2 i_1 \delta)$. In other words, e_2 conflicts with e_1 if the result of i_1, i_2, or any future invocation could differ depending on the order in which i_1 and i_2 are executed.

The notions of conflict and C-histories as defined above are quite general. When applied to a system of single-valued objects with only **read** and **write** operations, they reduce to the corresponding definitions in database theory. The definition of conflict takes into account the semantics of operations and the arguments for each invocation by being defined in terms of $LegalResponses_O$, which in turn is defined in terms of SP_O. SP_O encodes the semantics of all the operations of O. The definition also accounts for the current state of objects because only extensions of *before(e)* are considered. Knowledge of the future behavior of transactions is included because extensions can be formed only by including transactions from the set $TRANS$. Since the only orderings that must necessarily be present in a C-history are those between conflicting invocations, our definition of C-schedulers includes schedulers that may use high levels of knowledge to avoid ordering invocations. Hence, the restriction to C-schedulers is not a major one.

CHAPTER 4

Schemas

4.1. Introduction

The specification of a distributed object describes its behavior from the point of view of external effects. In this chapter, we consider the internal implementation of a distributed object; that is, how invocations at different sites are coordinated to provide the behavior required at the external interface of the object. There are several ways in which an object can be implemented, while still meeting its external specification. We first describe two possibilities: a *centralized implementation* and a *replicated implementation*. We then discuss why they are inefficient and lay the groundwork for a more efficient implementation.

4.2. Two possible implementations

The centralized implementation is similar to the method described in [44]. One of the sites where the object is accessible is chosen as the "master," while the other sites are "slaves." All invocations are executed sequentially by the master. Invocations scheduled at the slaves are passed on to the master for execution. The results of such invocations are sent from the master back to the slaves, which give the result to the user. Such a centralized implementation makes sense if the slaves are sites with little or no processing capacity. This is true, for example, with bank teller machines connected to a central computer.

If, on the other hand, all sites have comparable processing capacity (and the results in this work are primarily aimed at such systems), a replicated implementation is possible. Here, a copy of the object data and the definitions of the operations are placed at each site where the object is accessible. When an invocation is scheduled for execution, the site at which it is scheduled (i.e. the *local site*) informs the other sites where the object is accessible (the *remote sites*) of the execution. All the sites execute the operation in question, each using its own copy of data.[1] The local site returns the result to the user. This implementation requires some mechanism to ensure that invocations are executed in the same order at all sites, otherwise a copy of the data could become inconsistent with other copies. Because of such inconsistencies, a site may return results that are not permitted by the specification. This is described in the following example.

Consider an integer object x such that $Accessible_x = \{s, t\}$. Assume that following totally ordered sequence of invocations occurs: [x.**write**(2), s] [x.**write**(3), s] [x.**read**(), s] [x.**read**(), t]. If x behaves according to its specification, the result for both the **read** invocations should be **ok**(3), because the invocation **write**(3) followed the invocation **write**(2). Assume that the write operations on the copy of x at site s occur in the order above, but the writes are erroneously performed in the opposite order at site t, that is, **write**(3) occurs before **write**(2). When the invocation [x.**read**(), t] is performed on the copy at t, the result will be **ok**(2), which is incorrect.

[1]An optimization is possible in the case of operations that do not change the state of an object, e.g. a **read** operation on an integer. These can be performed at any one site without informing the others of it.

One way to avoid this problem in a replicated implementation is to acquire a "lock" at all sites before any operation is executed. A lock acquired at a site is released when the execution is completed at that site. A lock cannot be acquired a site if it is already acquired there for another invocation; the later invocation is delayed until the earlier one completes and the lock is released. This scheme ensures that invocations are performed in the same order at every site. However, deadlock may occur if an attempt is made to acquire locks for more than one invocation simultaneously – one invocation may acquire the lock at some sites, while another invocation acquires it at the other sites. The scheme can be modified to use a deadlock-free protocol to acquire locks, or to detect when deadlock occurs and take corrective action.

The advantage of a replicated implementation is that even if one or more of the sites at which an object is accessible fail, the operational sites can continue to accept and process invocations because an up-to-date copy of the object's data is available at the operational sites. This is not true of the centralized implementation, where if the master fails, the object cannot process invocations scheduled at the slaves.

The disadvantage of both the centralized and the replicated implementations given is that they are essentially synchronous. Both execute operations sequentially, the centralized implementation doing so at the master, while in the replicated implementation all sites operate in tandem. A synchronous implementation has the following drawbacks. Every invocation requires communication between sites. This increases the number of messages being sent in the system. This message transmission also introduces latency when operations are performed, because a user invoking an object must wait for a message to be sent between sites before receiving a result (Section 1.2).

Clearly, it is desirable to reduce the number of messages sent, to cut down latency and to perform operations in parallel when possible. To do so, it is necessary to decouple the execution of an operation at the local site from its execution at remote sites. The local site can then execute an operation and return its result without waiting for remote sites. This eliminates latency. In [17, 45], this issue is discussed in the context of database systems. Decoupling remote execution also means that while an operation is being executed at a local site, other operations can be executing at the remote sites, thus taking advantage of the parallelism in the system. Of course, this decoupling must be done in a way that does not result in inconsistency as exhibited in our earlier example. In Chapter 5, we present two implementations that achieve these aims. We now model the internal implementation of a distributed object by means of *schemas* and derive a result that will be used as a basis of those implementations.

4.3. Schemas

Except in the most trivial objects (e.g. one whose operations always return the same result), it is not possible to totally decouple events at one site from those at others. To provide a correct response, a local site must have information about invocations that have been scheduled at other sites. However, it is not always necessary to have knowledge of *all* such invocations. For example, consider an object of type **integer**. A local site can provide a correct result for a **read** invocation if it has knowledge of all **write** invocations; it does not need information about other **read** invocations. In general, a site in a non-synchronous implementation responds to an invocation based on *partial* information about the history of an object.

We call the partial history upon which an implementation bases its response to an invocation ι the *perspective* for ι. In both the centralized and the replicated implementations described earlier, for example, the perspective for any invocation ι is the entire history of the object. For an integer object implemented as described above, the perspective for any **read** invocation is the part of the history containing all the **write** invocations. In this work, we are not concerned with how a particular implementation may be expressed or described. We observe only that given an implementation, it is always possible to define the perspective for any invocation ι, even if only by exhaustive enumeration. Thus we can model an implementation in terms of perspectives. We formalize these ideas below.

An implementation of a distributed object is modeled by a *schema*. A schema is a function that when given an object history $H_O = [I_O, \rightarrow_O]$ and an invocation $\iota \in last(H_O)$ gives an *ι-history*. An ι-history is a history $H'_O = [I'_O, \rightarrow'_O]$, where I'_O is a subset of I_O that contains ι. The partial order $\rightarrow'_O$ is formed by taking all the elements in $\rightarrow_O$ that are relations between invocations in I'_O. In other words, $\rightarrow'_O = \{(j, k) \mid j, k \in I'_O \text{ and } (j, k) \in \rightarrow_O\}$. The ι-history given by a schema defines the perspective for an invocation ι in the corresponding implementation.

4.3.1. Correctness of a schema

We now discuss what it means for a schema S to be correct. For all histories and for all invocations ι, the result returned based on the ι-history specified by S should be the same as one that would be returned based on the entire history. Formally, let H_O be any history for object O, and let ι be any invocation in $last(H_O)$. Let D_O be any development consistent with H_O. Let H'_O be the ι-history defined by schema S and let D'_O be any development consistent with H'_O. S is a correct schema if $LegalResponses_O(D'_O) \subseteq LegalResponses_O(D_O)$.

4.3.2. Correctness with respect to a class of system histories

We have defined what it means for a schema to be correct with respect to all possible histories H_O. However, if it is known that all system histories belong to a particular class and that all object histories are projections of histories from this class, then a schema need be correct only with respect to such histories. It is often possible to optimize object implementations in such a way that they are efficient for histories from a particular class, but may be inefficient or even incorrect when histories are not from this class. In particular, we are interested in implementations that are efficient with respect to projections of C-histories. It is straightforward to extend the definition of correctness of a schema to give the definition of correctness with respect to a particular class X of system histories. We merely replace "let H_O be any history for object O" in the definition of correctness by "let H_O be any history that is a projection of a X-history on object O."

4.3.3. C-schemas

Let $H_O = [I_O, \rightarrow_O]$ be a projection of a C-history on O and let $\iota = [O.\mathbf{op}(), s]$ belong to $last(H_O)$. A *C-schema* is one in which the ι-history $H'_O = [I'_O, \rightarrow'_O]$ satisfies the following conditions.

1. $\iota \in I'_O$.

2. If $j \in I'_O$ and $k \rightarrow_O j$, then $k \in I'_O$ and $k \rightarrow'_O j$.

In other words, the ι-history specified by a C-schema contains all invocations on O that the scheduler orders before ι, and maybe other invocations. If the ι-history contains invocations that are not ordered relative to ι, then it also contains all invocations ordered before these invocations. We show below that C-schemas are correct with respect to C-histories. A C-schema will be used in Chapter 5 as the basis of an efficient implementation of distributed objects.

4.3.4. Correctness of a C-schema with respect to C-histories

We now show that if an object responds to each invocation ι based on an ι-history specified by a C-schema, the results returned are consistent with a response based on the entire object history, given that the system history is a C-history.

Let H_O be a projection of a C-history on O and let $\iota = [O.\mathbf{op}(), s]$ belong to $last(H_O)$. Let D_O be any development consistent with H_O. Let H'_O be an ι-history resulting from a C-schema, and let D'_O be any development consistent with H'_O. To prove that a C-schema is correct with respect to C-histories, we must show that $LegalResponses_O(\iota, D'_O) \subseteq LegalResponses_O(\iota, D_O)$.

The proof is as follows. We construct a sequence of developments $D_O = D^0$, D^1, D^2, $\cdots$, D^{last} such that D'_O is a prefix of D^{last}. Each development D^k is consistent with H_O. For all $k > 0$, D^k is obtained from D^{k-1} by switching the order of two invocations, but maintaining the property that $LegalResponses_O(D^k) = LegalResponses_O(D^{k-1})$. Hence, $LegalResponses_O(D^0) = LegalResponses_O(D^{last})$. Since $D_O = D^0$, $LegalResponses_O(D_O) = LegalResponses_O(D^{last})$, and hence $LegalResponses_O(\iota,\ D^0) = LegalResponses_O(\iota,\ D^{last})$. Because D'_O is a prefix of D^{last} and specifications are complete and prefix closed, it follows that $LegalResponses_O(\iota,\ D_O) = LegalResponses_O(\iota,\ D'_O)$.

The algorithm used to construct the sequence of developments is given below. A proof that the algorithm terminates and a proof of its correctness follow. We use $\alpha_1\alpha_2 \cdots \alpha_m$ to represent D'_O and $\beta_1 \cdots \beta_n$ to represent D^k for the current value of k (each α_* or β_* represents an invocation on O). Note that $n \geq m$ because all D^k are consistent with H_O, and H_O contains all the invocations in H'_O.

4.3.4.1. The algorithm

(1) Let $k = 0$ and $D^k = D_O$.

(2) Let p be the largest integer $0 \leq p \leq m$ such that $\alpha_1 \cdots \alpha_p = \beta_1 \cdots \beta_p$. In other words, p is the length of the longest common prefix of D'_O and D^k.

If $p = m$, D'_O is a prefix of D^k and the algorithm terminates.

(3) If $p \neq m$, consider the invocation α_{p+1}.

α_{p+1} must occur in D^k because D^k contains all the invocations in I_O, and I_O includes all the invocations in I'_O and hence in D'_O.

Let β_l be the same invocation as α_{p+1}.

Note that $l > p + 1$, otherwise the longest common prefix could have been extended to $p + 1$.

(4) The next development D^{k+1} is $\beta_1 \cdots \beta_{l-2}\beta_l\beta_{l-1}\beta_{l+1} \cdots \beta_n$; that is, D^{k+1} is the same as D^k but with the positions of β_{l-1} and β_l interchanged. We show below that $LegalResponses_O(D^{k+1}) = LegalResponses_O(D^k)$.

(5) Set k to the value of $k + 1$ and go to step (2).

Figure 4.1 shows one step of the algorithm with $k = 0$ and $p = 3$.

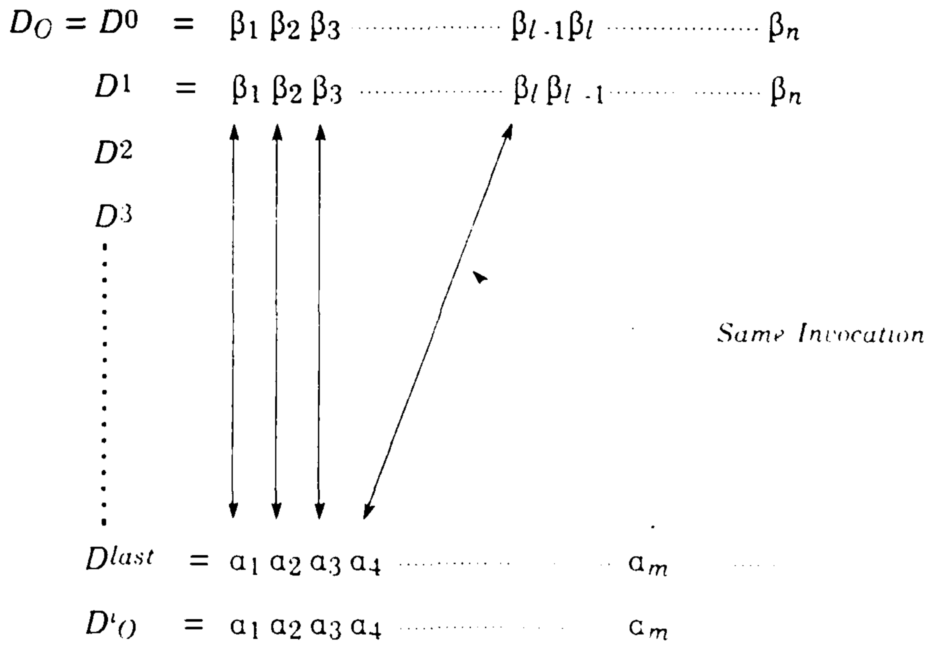
$D_O = D^0 = \beta_1 \beta_2 \beta_3 \cdots \beta_{l-1}\beta_l \cdots \beta_n$
$D^1 = \beta_1 \beta_2 \beta_3 \cdots \beta_l \beta_{l-1} \cdots \beta_n$
D^2
D^3
Same Invocation
$D^{last} = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \cdots \alpha_m$
$D'_O = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \cdots \alpha_m$

Figure 4.1. A step in the algorithm

4.3.4.2. Proof of termination

Each iteration moves the position of β_l (corresponding to α_{p+1} in D'_O) one position to the left in D^k to give D^{k+1}. Hence after $l - (p + 1)$ iterations, this invocation will reach position $p + 1$ in D^k. The next iteration will increase p, the length of the longest common prefix between D'_O and D^k, to at least $p + 1$ and repeat the process. Thus the value of p continues to increase as the algorithm progresses. Eventually its value will become equal to m and the algorithm will terminate.

4.3.4.3. Proof that all developments D^k are consistent with H_O

This is proved by induction on k. The base case, $k = 0$, follows immediately since $D^0 = D_O$, which is consistent with H_O.

For the inductive step, assume that D^k is consistent with H_O. Since $l > p + 1$ (see note in Step 3), the invocation β_{l-1} does not occur in $\beta_1 \cdots \beta_p$ and because $\alpha_1 \cdots \alpha_p = \beta_1 \cdots \beta_p$, it is not one of the first p invocations in D'_O either. The next invocation in D'_O, α_{p+1}, is the same as β_l (Step 3). Hence the invocation β_{l-1} must occur after α_{p+1} (viz. β_l) in D'_O or not occur in D'_O at all. We consider these two possibilities separately below and show that in either case, β_l and β_{l-1} are unordered relative to each other in H_O.

If the invocation β_{l-1} occurs in D'_O, it occurs after β_l. If β_{l-1} and β_l were ordered relative to each other in H_O, it follows from the definition of a schema that they would be ordered the same way in H'_O. However, β_{l-1} occurs before β_l in D^k, which is consistent with H_O, and after β_l in D'_O, which is consistent with H'_O. It follows, then, that β_{l-1} and β_l must be unordered in H_O.

The other possibility is that β_{i-1} does not occur in D'_O. The definition of C-schema implies that if an invocation $j \in I'_O$, then all invocations k such that $k \rightarrow_O j$ also belong to I'_O. The invocation β_i occurs in D'_O. Hence, if β_{i-1} is not present in D'_O, it follows that β_{i-1} and β_i are unordered relative to each other in H_O, otherwise H'_O could not be based on a C-schema.

By the inductive hypothesis, D^k is consistent with H_O. The order of all invocations in D^{k-1} is the same as in D^k, except for β_{i-1} and β_i. But these invocations are unordered in H_O. Hence, D^{k-1} is consistent with H_O.

4.3.4.4. Proof that $LegalResponses_O(D^{k-1}) = LegalResponses_O(D^k)$

D^{k+1} and D^k differ in the order of β_{i-1} and β_i, and as shown above these invocations are unordered in H_O. There are two possible cases: $time(\beta_{i-1}) < time(\beta_i)$ or $time(\beta_{i-1}) > time(\beta_i)$. The proof in both cases is similar and will be developed in parallel, with the second case in square brackets.

Assume $time(\beta_{i-1}) < time(\beta_i)$ [resp. $time(\beta_i) < time(\beta_{i-1})$], that is $\beta_{i-1} \in before(\beta_i)$ [$\beta_i \in before(\beta_{i-1})$]. Now H_O is an extension of $before(\beta_{i-1})$ [$before(\beta_i)$]. Since H_O is a projection of a C-history and β_{i-1} and β_i are unordered in it, it follows that β_i does not conflict with β_{i-1} [β_{i-1} does not conflict with β_i]. D^k is a development consistent with H_O of the form $\gamma\beta_{i-1}\beta_i\delta$. The absence of a conflict means that $LegalResponses_O(\gamma\beta_{i-1}\beta_i\delta) = LegalResponses_O(\gamma\beta_i\beta_{i-1}\delta)$, which gives the result.

4.4. Summary

In this chapter, we introduced the concept of schemas and described what it means for a schema to be correct. We defined C-schemas and proved that they are correct with respect to C-histories. In the next chapter, we use a C-schema to develop an efficient implementation of distributed objects.

CHAPTER 5

Implementation

5.1. Introduction

In this chapter, we use a C-schema to develop two implementations for distributed objects – a *piggybacked implementation* and a *concurrent* one. In both implementations, the result of an invocation is returned as soon as it is executed at the local site, without requiring the user to wait for a communication with remote sites. The latency described in Section 1.2 is thus eliminated. The information sent between sites by the piggybacked implementation is *added to messages* already being used by the scheduler for synchronization. This is an immediate advantage in systems where the number of messages in the system is a dominant factor in system performance, because the piggybacked implementation requires no additional messages.[1] The concurrent implementation is a modification of the piggybacked one in which information about invocations is exchanged between sites in parallel with the execution of other operations. This implementation is intended for use in systems where the number of messages sent is not a major constraint. Its advantage is that it leads to a more even distribution of work than does the piggybacked implementation.

[1]In practice it is not possible to piggyback arbitrary amounts of information on existing messages without causing them to be fragmented into a number of smaller packets for transmission. Thus there could be *some* additional cost. We show later how to keep the amount of information piggybacked small.

5.2. The piggybacked implementation

In the piggybacked implementation of an object O, a *component* is placed at each of the sites in $Accessible_O$. Each component has a copy of $DATA_O$ and the definitions of operations in OP_O, and can independently perform operations on its copy of data. When an invocation is scheduled at a site, the component at that site performs the requested operation using its copy of data, and returns the result immediately. It then instructs each of the other components to perform the same operation on their copies. Components are informed of invocations scheduled at other components in such a way that the resulting implementation corresponds to a C-schema. The correctness of a C-schema (Section 4.3.4) implies that the results returned by the individual components are consistent with the specification of the object as a whole.

When an invocation is scheduled at a site, it is assigned a *timestamp*. Timestamps have the property that of any two invocations scheduled at the same site, the one scheduled earlier has a smaller timestamp. The timestamp of an invocation, with the site name appended, is called the *invocation-ID*. Note that timestamps and invocation-ID's can be generated locally at each site, and require no global synchronization. A *notification* for a particular invocation is a record containing the invocation-ID, the name of the object, the arguments for the invocation, and the name of a destination site. Notifications are sent from the component where an invocation is scheduled to each of the other components. When a component receives a notification, it executes the named operation on its copy of data.

In the piggybacked implementation, copies of notifications created at a site s are ordered according to timestamp and are piggybacked on *all* subsequent synchronization messages sent from s. When a synchronization message arrives at a site t, the piggybacked notifications are first processed in order. Then the synchronization message itself is acted upon. For each piggybacked notification, the following occurs. If t is the destination site for the notification, the notification is delivered to the component at t, which executes the named operation. Otherwise, a copy of the notification is saved, and is piggybacked on all synchronization messages that are subsequently sent from t. Thus, copies of a notification may travel from site to site and may reach its destination by many different paths. Below, we show how the invocation-ID's are used to ignore all but the first copy that arrives at a site, and to purge copies of notifications from the system once a copy has reached its destination. The method used is similar to the algorithm in [47].

5.2.1. Transmission of notifications

The algorithm followed at a site s to distribute notifications is shown in Figure 5.1. Each site s maintains a buffer $Outgoing_s$ of outgoing notifications. $Outgoing_s$ contains notifications originating at s as well as copies that arrive at s *en route* to other destinations. A copy of a notification remains in $Outgoing_s$, and continues to be piggybacked on all outgoing synchronization messages, until site s learns that the destination has received a copy.

Observe that if a notification n_1 created at s carries a smaller timestamp than another notification n_2 also created at s, then a copy of n_1 must be received at any other site t before the first copy of n_2 is received there. This is because if n_1 does

Whenever a synchronization message m is being sent from site s to site t:

- Piggyback on m a copy of all notifications in $Outgoing_s$, in order.
- Piggyback the values of $LargestSeen_s$ and $TheirLargest_s$.

When a synchronization message is received from site t:

- For each site v, accept all piggybacked notifications originating from v whose timestamps are greater than $LargestSeen_s[v]$, and set the value of $LargestSeen_s[v]$ to the largest such timestamp.
- Process, in order, all notifications with destination s, and append all other notifications to $Outgoing_s$, preserving their order.
- Set the values of $TheirLargest_s[t][v]$ to the piggybacked values of $LargestSeen_t[v]$.
- Set the value of $TheirLargest_s[v][w]$ to the larger of $TheirLargest_s[v][w]$ and the piggybacked value of $TheirLargest_t[v][w]$.
- Delete from $Outgoing_s$ all notifications from site w to site v with timestamps smaller than or equal to $TheirLargest_s[v][w]$.

Figure 5.1. Piggybacked implementation, as followed by site s.

not arrive first at t by another path, then a copy of n_1 will be piggybacked on the same path of synchronization messages as n_2, and will be ordered before n_2. Hence, if each site s keeps track of the largest timestamp it has observed on a notification originating from each other site t, it can ignore all notifications with smaller timestamps, as it must have already received a copy.

At each site s, the array element $LargestSeen_s[v]$ records the largest timestamp that s has observed on a notification originating from v. Notifications with smaller timestamps are ignored by s. The array element $TheirLargest_s[v][w]$

records at s the value of $LargestSeen_v[w]$ at the time of the last message that v sent to s. Site s deletes from $Outgoing_s$ any notification originating at w with destination v that carries a smaller timestamp than $TheirLargest_s[v][w]$, because v must have already received a copy. For the case $w = s$, this means that s deletes copies of any notification created at s that it knows to have been received at the destination v.

5.2.2. Correctness

We now show that the piggybacked implementation corresponds to a C-schema. A component executes operations in the order in which it creates notifications or first receives them from other components. We must show that whenever an invocation i is executed at a component, the invocations executed at that component form a development that is consistent with the i-history specified by a C-schema.[2] In other words, we must show that if we have two invocations i_1 and i_2 on the same object such that $i_1 \rightarrow_{SYS} i_2$, then i_1 is executed before i_2 at all the components. In Section 2.6, we observed that if $i_1 \rightarrow_{SYS} i_2$, then there is a message path from i_1 to i_2; that is, there is a sequence of messages from s, where i_1 is scheduled to t, where i_2 is scheduled. The first message in this sequence is sent from s after i_1 is performed there, and the last one arrives at t before i_2 is performed. Each intermediate message is sent from the destination of the previous one in the sequence, and the sending occurs after the receipt of the previous message. Because such a message path exists, a copy of the notification for i_1 will be

[2]Recall that a development is simply a sequence representing the order in which invocations are processed at a component.

piggybacked along this path and will arrive at t before i_2 occurs there. (For the case $s = t$, it is convenient to think of the scheduler as ordering consecutive invocations at the same site by sending a message from that site to itself, though this need not actually be implemented.) Hence, i_1 will be performed before i_2 at site t.

We have seen that a copy of the notification for i_1 arrives at t before i_2 occurs there. At the other sites, if the notification for i_1 does not arrive earlier by another path, a copy will be piggybacked on the same sequence of messages that the notification for i_2 is piggybacked upon, and will be ordered before the notification for i_2. Hence, every other component will also execute i_1 before i_2.

This shows that the order in which invocations are performed at each component is consistent with a C-schema. The piggybacked implementation is hence correct when a C-scheduler is used.

5.2.3. Optimizations

A number of optimizations are possible. An obvious drawback is that notifications are piggybacked on synchronization messages that might never lead to their intended destination. This could be avoided if the scheduler indicates which object a particular synchronization message refers to. For example, if a lock-based scheduling method is being used, the objects corresponding to a particular lock acquisition or release message are known. Notifications could then be piggybacked only on those synchronization messages that refer to the objects in question.

Another optimization, which is simple to implement, is to not piggyback on a message to a site t those notifications in $Outgoing_s$ that have already been piggybacked on an earlier message to t, even if their timestamps are larger than $TheirLargest_s[v][w]$.

The sizes of the buffers $Outgoing_s$ as well as the number of notifications piggybacked can also be controlled by periodically broadcasting $LastSeen_s$ to other sites. This enables them to promptly update their values of $TheirLargest$ and discard copies of notifications that have already reached their destinations. If this is carried out frequently enough, the only notifications that should be in the buffers are those in transit; that is, those for which the destination has not received a copy. Broadcasting the values of $LastSeen$, however, adds to the message traffic in the system.

5.2.4. Discussion

Let us consider the features of the piggybacked implementation. First, a component can return the results of an invocation when it is carried out locally, without having to wait for the other components to be notified of it. This eliminates latency. Second, no additional messages are required for communication between components: all the necessary information is piggybacked on messages already used by the scheduler for synchronization. Third, the implementation is independent of the algorithm used by the scheduler, provided that it falls into the class of C-schedulers. Even if the scheduler uses information like the semantics of operations, the current state of objects, or the arguments for a particular invocation, the messages it sends are sufficient for the implementation to be correct.

The piggybacked implementation shows that if data are to be accessed from multiple sites in a distributed system, they can be replicated at these sites in a way that requires no extra cost in terms of latency or number of messages. It must be pointed out that a scheduler that permits access to an object from multiple sites may have to use a large number of messages for synchronization. This, however, is a cost resulting from permitting distributed access to data, and is independent of whether data are replicated or not. The piggybacked implementation demonstrates that the advantages of increased availability in the presence of failures and the benefits of placing a copy of data at sites easily accessible by users need be balanced only against the costs of storing multiple copies of data and of having to process more than one copy. These costs are unavoidable if data are replicated.

5.3. The concurrent implementation

The piggybacked implementation has the following property. A synchronization message arriving at a site to schedule an invocation ι for execution may have a large number of notifications piggybacked on it. One result is that synchronization messages may become very large. Moreover, the execution of ι will be delayed until all the piggybacked notifications have been processed and all invocations ahead of ι have been performed. Note that this delay is different from the latency described earlier, which was a wait for a message to be transmitted to a remote site and for a reply to be received. The delay described here is a wait for local processing to take place, which usually takes less time than that required for message transmission. Nevertheless, this bursty pattern of execution, where a large number of operations may have to be executed when a synchronization message arrives and relatively little is done at other times, could lead to inefficient use of computational resources. In systems where this is a performance issue, and where the number of messages in the system is not a major constraint, the concurrent implementation gives better performance.

In this variation, an invocation *descriptor* is piggybacked on synchronization messages. A descriptor for an invocation consists of the invocation-ID and a destination site. Notifications are transmitted directly to the destinations using an *atomic broadcast*, which has the following properties.

1. The data broadcast are either received at all operational destinations, or at none at all, even if site failures occur during the broadcast. Moreover, if an atomic broadcast B_2 is initiated from a site after another atomic broadcast B_1 from the same site, and if the data broadcast by B_2 are received at its destinations, then the data broadcast by B_1 are also received at its destinations.

2. If two atomic broadcasts made from the same site have destinations in common, the data are received at overlapping destinations in the order that the broadcasts were initiated.

3. If the data from an atomic broadcast B_1 is received at a site before an atomic broadcast B_2 is initiated from that site, then the data from B_1 are received before the data from B_2 at any overlapping destination.

A number of protocols have been proposed for implementing broadcasts with these and similar properties [8, 14, 15, 41]. In [8], we describe a communication sub-system that provides an atomic broadcast as a primitive operation. We denote the initiation of the atomic broadcast for invocation i as $AtBcast(i)$.

The concurrent implementation may be described in terms of two rules: a *broadcast ordering rule*, which governs the order in which notifications are transmitted, and a *blocking rule*, which specifies when the execution of certain operations must wait for others. As in the piggybacked implementation, the result of an invocation is returned once it has been completed at the local site. The atomic broadcast of its notifications may be initiated after an arbitrary amount of time, but must follow the broadcast ordering rule: if an invocation i_1 is scheduled before another invocation i_2 at the same site, then $AtBcast(i_1)$ occurs before $AtBcast(i_2)$. This condition can be enforced locally. It is possible, as an optimization, to package more than one notification into the same atomic broadcast, provided their order is observed at the destinations.

At a destination, operations are performed in the order that notifications are received. Note that the concurrent implementation permits $AtBcast(i)$ to be initiated any time after i is scheduled, provided the broadcast ordering rule is not violated. Thus, notifications can be transmitted in such a way that they arrive at their destinations spaced out over time, instead of bunched up behind synchronization messages, as in the piggybacked implementation. This distributes the load arising from executing operations, leading to fewer potential bottlenecks in the utilization of system resources. The trade-off is the increased number of messages in the system.

The blocking rule remains to be described. The piggybacking of invocation descriptors on synchronization messages ensures that if i_1 and i_2 are two invocations on the same object scheduled at sites s and t respectively, and if $i_1 \rightarrow_{SYS} i_2$, then a copy of the piggybacked descriptor for i_1 is received at t before i_2 is performed there (Section 5.2.2). However, because the transmission of notifications may be delayed arbitrarily, the notification for i_1 may have not yet arrived at t, and the corresponding operation not executed. The blocking rule is enforced if an invocation descriptor for, say, i_1 has been received at a site but the corresponding notification has not arrived. The rule states that in this situation, if any invocation i_2 is scheduled at that site after the receipt of the descriptor for i_1, then the execution of i_2 is blocked until the notification for i_1 has been received and the corresponding operation performed. Note that this kind of blocking occurs only if the transmission of a notification is unduly delayed. If notifications are broadcast promptly after invocations are scheduled, this situation should be infrequent.

5.3.1. Correctness

To show that the concurrent implementation corresponds to a C-schema, we must show that if we have two invocations i_1 and i_2 on the same object and if $i_1 \rightarrow_{SYS} i_2$, then i_1 is performed before i_2 at all components.

Let i_1 and i_2 be scheduled at sites s and t respectively. There is a path of synchronization messages from s to t such that a piggybacked descriptor for i_1 will arrive at t before i_2 is performed there. If the notification for i_1 has not already arrived at t, the blocking rule ensures that the execution of i_2 will be delayed until the notification for i_1 arrives and i_1 is performed at t. Hence, i_1 occurs before i_2 at the component at t.

To show that i_1 is performed before i_2 by the other components as well, we consider two cases. If $s = t$, then the broadcast ordering constraint ensures that $AtBcast(i_1)$ is initiated before $AtBcast(i_2)$. The properties of an atomic broadcast ensure that the notification for i_1 arrives before that for i_2 at all destinations, so i_1

occurs before i_2 at all components. If $s \neq t$, we know from the argument above that the notification for i_1 arrives at t before $AtBcast(i_2)$ is initiated there. Again, the properties of atomic broadcasts ensure that the notification for i_1 reaches all destinations before that for i_2. Hence, i_1 is performed before i_2 everywhere.

It follows that the concurrent implementation is consistent with a C-schema, and is hence correct when a C-scheduler is used.

5.3.2. Discussion

The concurrent implementation, like the piggybacked implementation, eliminates latency. It spreads out the execution of remote operations over time, thus leading to better utilization of system resources. The decision as to when to perform an atomic broadcast to distribute notifications is left unspecified. This opens the possibility of a *message scheduler* being used to make this decision based on the current load on the network, the state of the message transmission buffers, and other factors. Such a message scheduler could have a significant influence on the performance of the concurrent implementation.

The concurrent implementation of distributed objects makes a high level of concurrency possible in replicated distributed systems. If implemented at a sufficiently low level in the system, this concurrency may be obtained in a way that is transparent to high level application programs. Thus, applications could perform very efficiently without requiring complex programming to achieve this level of performance. This idea is being explored in the *ISIS* system, currently being developed at Cornell [6, 7, 9, 10].

CHAPTER 6

Failures

6.1. Introduction

One of the reasons for replicating data is to keep objects accessible when sites fail. In this chapter, we discuss how the implementations of distributed objects given in Chapter 4 may be integrated with a failure handling mechanism. We first consider a roll-back mechanism, where a failure causes transactions to be undone and re-executed using another copy of the data. We then discuss a roll-forward mechanism, where transactions in progress at a failed site are completed by another site that takes over from it.

The type of failures we consider are *fail-stop* site failures [39]: a site fails by halting all execution and sends no more messages; its failure is detectable by every other site in the system. Any information about the current state of data stored at a site is lost when it fails; when a site recovers, it copies up-to-date information from operational sites. We do not consider failures where a site takes incorrect actions while remaining operational or where a site sends out spurious messages. This precludes network partitioning, where a set of sites remain operational, but are unable to communicate with the other sites. The communication medium is assumed to be error-free.

The abstraction of fail-stop sites can be implemented to a high degree of accuracy in software [40]. A software layer, called the *failure detector*, monitors the sites and the communication medium. It implements fail-stop sites by shutting down any site suspected of malfunctioning, cutting off communication to and from the site, and announcing to the rest of the system that the site has failed. In the event of network partitioning, the failure detector may block activities until the partition is resolved.

6.2. Roll-back recovery

Many transaction-based systems handle failures by undoing the effects of ongoing transactions that have accessed failed sites. These transactions are re-executed when the failed site recovers, or if the data are replicated, they are re-executed using another copy of the data. In such systems, the changes made by a transaction to the data of an object are not made permanent until the transaction terminates, at which time it may *commit* or *abort*. A commit represents normal termination, while an abort results in each object being restored to a state in which it would have been had the transaction not executed. Thus, when a site fails, the failure handler simply aborts all ongoing transactions that have accessed the failed site. These transactions are later re-executed.

Hadzilacos has studied the scheduling problem in an environment where transactions may be aborted as a result of system failures, and discusses restrictions on the class of serializable histories that are necessary or useful in this setting [23]. He observes that when failures can occur, the notion of serializability should be defined in terms of *committed* transactions; that is, it must account for the fact that a system failure may occur at any time, leading to an abort of one or more of the uncommitted transactions. The essential condition is that the system history must be *commit serializable*: the history formed by including only the committed transactions should be serializable, and this must be true of any prefix of the system history as well. It is also shown that any history belonging to the class *CPSR* (corresponding to *C*-histories in our model) is commit serializable.

Hadzilacos also discusses other properties desirable of a scheduler operating in a system where transactions may be aborted. A system history is said to be *recoverable* if the abortion of any ongoing transaction does not invalidate the results of a committed transaction. If a scheduler generates system histories that are not recoverable, an abort could require that a committed transaction be undone, which is undesirable and may even be impossible. Finally, he strengthens the notion of recoverability to *strictness*. Strict histories have two advantages. First, they avoid the problem of *cascading aborts* – the situation where the abort of one transaction requires that other active transactions be aborted as well, because their actions depended on some of the actions of the aborted transaction. While cascading aborts does not compromise correctness, maintaining information about the dependencies between transactions is a complex and inefficient task [20, 35]. The other advantage of strict histories is that a transaction can be aborted simply by restoring each object to the state that it was in at the time the transaction started executing, regardless of the actions of any concurrently executing transactions.

The results of Hadzilacos are a strong argument in favor of restricting the scheduler to produce only strict histories, if transactions can be aborted. Hadzilacos shows how to modify commonly used scheduling disciplines – two-phase locking, timestamp ordering, and serialization graph testing – so that they may be used in an environment where aborts may occur. The changes are minor, and ensure that only strict histories are generated. Since the result we proved in Chapter 4 holds for any C-scheduler, it remains valid when such changes are made.

We now outline how aborts can be included in the model of Chapter 2. Each object is considered to have two special operations: **begin** and **abort**. Invoking **abort**(Tid) restores the state of the object to the state that existed when **begin**(Tid) was invoked. Hence, if the scheduler generates only strict histories, a transaction may be aborted by invoking an **abort** operation for every object it accesses. To allow for this, the set of all possible transactions $TRANS$ is extended as follows. Each transaction T in $TRANS$ is preceded by an invocation of **begin** for each object that the transaction accesses. Further, for each transaction T in $TRANS$ and each invocation ι in T, a new transaction is added to $TRANS$, which invokes an **abort** operation after ι for every object that T accesses, and has no subsequent invocation. This reflects the fact that a transaction may be aborted at any time during its execution because of a failure. To the scheduler, which we assume generates only strict histories, **begin** and **abort** invocations appear as invocations of normal operations.

A roll-back recovery mechanism such as the one described above can be used in conjunction with a piggybacked or a concurrent implementation of objects in much the same way that it can be used with a synchronous implementation. It may appear that the implementation could be incorrect in the presence of failures because a piggybacked notification or descriptor could arrive at a site *en route* to another, and be lost at the intermediate site if the site fails. However, if an invocation is scheduled after a failure, there must be *some* path of synchronization messages that leads to it from every invocation ordered before it, otherwise the execution could be incorrect. The required notification or descriptor will reach its destination along this path, even if copies along other paths are lost.

A minor modification may be required, depending on how transactions are aborted. If **abort** operations are invoked independently from the normal scheduling mechanism, then it is possible for a notification from an aborted transaction to arrive at a component after the transaction has been aborted there. If the transaction ID's are included in notifications, such notifications can be detected and ignored. This problem does not arise if transaction aborts are integrated with the normal scheduling mechanism.

6.3. Roll-forward recovery

Aborting and re-executing transactions is just one way of handling failures. An alternative, especially when data are replicated, is to provide a roll-forward mechanism. By this we mean that transactions in progress at a site that fails are continued and completed by another site. Recovery schemes along these lines are presented in [10, 39]. The failure of a site is thus masked from a user (except perhaps as an increase in response time when a failure occurs), unless all the sites where a piece of data is replicated happen to fail. This, however, should occur relatively rarely.

When a roll-forward scheme is used in conjunction with a piggybacked or concurrent implementation, a number of problems arise. These are mainly due to the fact that the failure or recovery of a site can be detected at different times and in differing orders by other sites in the system. Despite this, sites must react consistently to the failure and provide correct responses for ongoing transactions. We illustrate the problems by means of a few examples.

Assume that one of the invocations of an ongoing transaction is scheduled at a site that subsequently fails. At another site, a component of the invoked object may observe any of the following outcomes.

1. The notification for the invocation arrives, and the failure is detected later.
2. The failure is first detected, and the notification arrives later.
3. The failure is detected, but the notification never arrives (because the site failed before sending the notification).

If a component detects the failure and wishes to continue execution of the transaction, it cannot tell whether a notification from the failed site will arrive or not (i.e. it cannot distinguish between cases 2 and 3). Also, if some of the components receive the notification before detecting the failure (case 1) and others receive it after detecting the failure (case 2), the actions they take must not lead to inconsistencies in copies of the object's data.

Another kind of problem is exemplified in the following scenario. Assume that it is agreed that the operational component with the lowest numbered ID takes over, completes and responds to invocations interrupted by a site failure. Let an invocation be scheduled at site s, which subsequently fails. The lowest numbered component, say at site t, completes the invocation. Assume that t also fails. If the component that now has the lowest ID detects the failure of site t before that of site s, it cannot know that t took over from s and completed the invocation. Hence, it would incorrectly re-execute the invocation and attempt to provide a second response.

The order in which recoveries, not just failures, are detected is also an issue. Recall that if a site fails, it loses all information about the current state of data. Hence, the usual action taken by a component when it recovers is to obtain an up-to-date copy of data from an operational component. For the recovered component to continue to behave correctly, it must receive notifications of all subsequent invocations scheduled at other components of the same object. However, if the other components detect the recovery at different times, they may neglect to send the recovered component notifications it should receive.

The problems described above arise when sites observe failures and recoveries at different times and in different orders. These situations are handled easily when a synchronous implementation is used for objects, because components coordinate their actions for every invocation. In a non-synchronous implementation, however, components are permitted to get out of step with each other, and the timing and order in which failures and recoveries are observed becomes an issue.

One solution is for the components of an object to run an agreement protocol each time a component detects a failure or recovery. They can then agree on the status of outstanding notifications and take a consistent set of actions to handle the failure or recovery. Another solution is to integrate the failure detector with the communication sub-system used by the scheduler. In [8], we describe such an approach. Our communication sub-system provides a set of broadcast primitives that order message delivery with respect to failure detection and recoveries. For example, it is not possible for data sent in the same broadcast to a set of sites to be received at some of the destinations before a failure is detected there, while arriving at other destinations after the same failure is detected at those sites. Moreover, each site observes failures and recoveries in a consistent order. The effect is to serialize failure and recovery decisions relative to other events in the system. With such a communication sub-system, problems such as those described earlier do not arise when a piggybacked or concurrent implementation is used. An additional advantage of using this communication sub-system is that it makes recovery protocols very simple to describe and program. These advantages have prompted us to re-implement the *ISIS* communication sub-system along these lines.

6.4. Conclusion

We have indicated how the piggybacked or concurrent implementations of distributed objects may be integrated with a roll-back or a roll-forward recovery mechanism. With the former, little modification is required. The latter requires a protocol that ensures that all sites have a consistent view of the timing and order of failures and recoveries. The existence of such protocols justifies our claim that the methods presented here are applicable in a wide range of fault-tolerant systems, and not just fault-intolerant ones that happen to be distributed.

CHAPTER 7

Performance

7.1. Introduction

The piggybacked or the concurrent implementation of a distributed object offers an advantage over a synchronous one because it does not incur a latency cost. To obtain a quantitative measure of this advantage, we compared the performance of a concurrent implementation with that of a synchronous one. Rather than building a system of distributed objects from scratch, we modified an already existing prototype of the *ISIS* system and carried out our measurements in this context. The figures we present should not be taken as an absolute measure of the performance obtainable from a concurrent implementation, because they include a large overhead arising from the *ISIS* system itself. They are, instead, intended to compare the performance of the concurrent and synchronous implementations, other things being equal. We are pleased to report that the concurrent implementation performed significantly better. When stripped of the overhead of the *ISIS* system, performance gains should be even higher.

7.2. The *ISIS* system

The *ISIS* system, under development at Cornell, aims to aid the construction of fault-tolerant software. It automatically converts fault-intolerant specifications of objects into fault-tolerant implementations. *ISIS* operates by transforming an object specification into a *resilient object*: an implementation of a distributed object

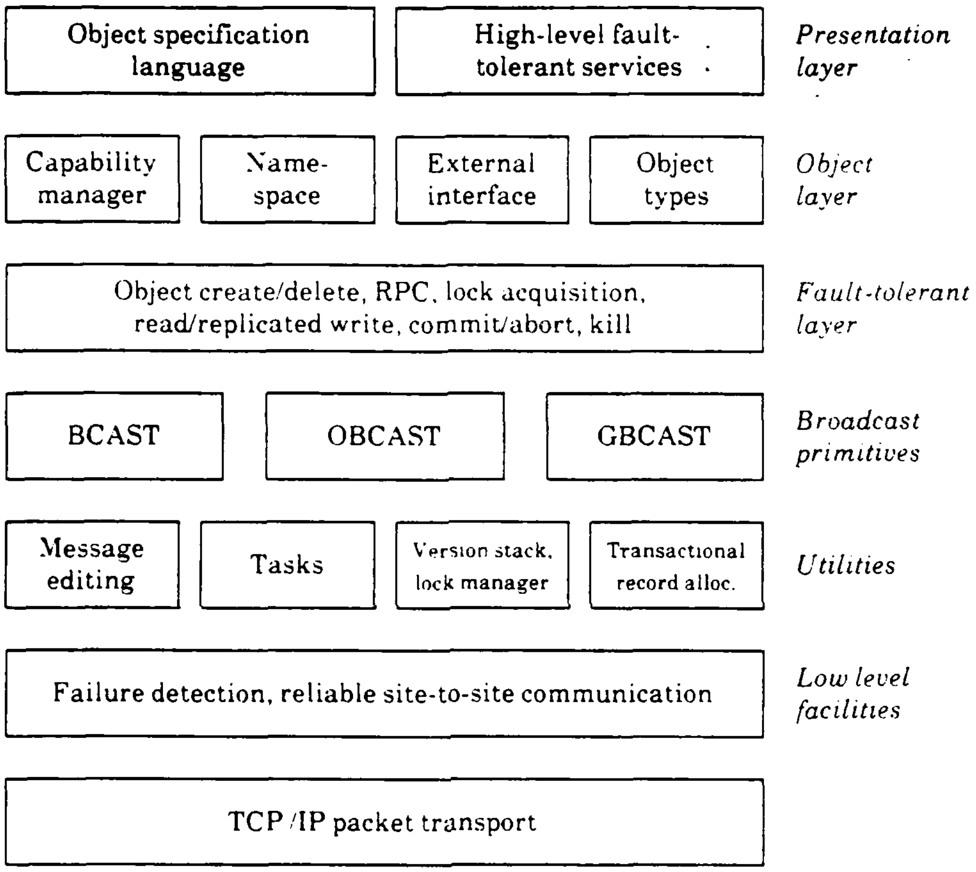
Object specification language
High-level fault-tolerant services
Presentation layer
Capability manager
Name-space
External interface
Object types
Object layer
Object create/delete, RPC, lock acquisition, read/replicated write, commit/abort, kill
Fault-tolerant layer
BCAST
OBCAST
GBCAST
Broadcast primitives
Message editing
Tasks
Version stack, lock manager
Transactional record alloc.
Utilities
Failure detection, reliable site-to-site communication
Low level facilities
TCP/IP packet transport

Figure 7.1. *ISIS* system architecture

that continues to accept and respond to invocations despite site failures. In a resilient object, data and programs are replicated at more than one site and actions at these sites are coordinated in a way that enables operational sites to continue to operate correctly even if one or more sites fail. The details of replication, the protocols used to maintain the consistency of replicated data, and the failure handling mechanisms are all hidden from a user of *ISIS*. A user specifies a resilient object in much the same way as he or she would program a fault-intolerant object. This enables a relatively inexpert user to build a fault-tolerant application. Details of the *ISIS* system can be found in [6, 7, 9, 10].

7.3. The *ISIS* prototype

A prototype of the *ISIS* system has been operational since January 1985. It runs on top of 4.2 UNIX on a cluster of 5 SUN 2/50 work-stations, interconnected by a 10 Mbit/s Ethernet. Figure 7.1 shows the hierarchical architecture of the *ISIS* system. The lowest level uses a site-to-site windowed acknowledgement protocol to provide sequenced, (almost) error-free message transmission. It detects site failures using timeouts and runs a protocol to ensure that all sites observe site failures and recoveries in a consistent order. On top of this layer are system utilities, which are used to implement broadcast primitives with different ordering properties. Resilient objects reside on top of this layer. Some of the system services like a name-space and a capability manager are built as resilient objects.

In UNIX, processes and inter-process communication are relatively expensive. Hence, a single system process is used at each site to handle functions common to all resilient objects. Also, only one process is used at each site to implement all objects of the same type, as in [34]. This process is called a type manager and multiplexes its time among all the objects of that type. A new process is created only when a new object type is installed at a site. Utilities to load and unload type managers are provided by the system process.

7.4. Performance measurements

To measure the performance of the concurrent implementation, the system was configured so that objects could function either in a synchronous or in a concurrent mode. In the synchronous mode, a message is transmitted to all components of the object for each operation (or sub-operation), and the next operation is executed only after an acknowledgement is received. This ensures a consistent order of execution at each component. In the concurrent mode, the method of Section 5.3 (the concurrent implementation) is used for all resilient objects. The implementation in *ISIS* differs from the method as presented here in one respect. Because of the way *ISIS* is constructed, each operation is broken up into read and write sub-operations before being executed. This results in more than one notification being sent for each invocation. The additional message overhead makes the performance of the concurrent implementation appear poorer than it should be. Table 7.1 presents some general performance measurements for the system, to put the later figures for object implementations into perspective.

Table 7.1. General performance figures

Broadcast	(delay till reception)	10 ms
RPC to object (local site)	(delay till task begins)	30 ms
RPC to object (remote site)	(delay till receipt of acknowledgement)	40 ms

In Table 7.2, we see the results of measurements made on four objects: a file with **read** and **write** operations, a directory with **bind** and **lookup** operations, a stack with **push** and **pop** operations, and an indexed file with **add** and **lookup** operations. In each case, we measured the average time to execute 25 invocations of the designated operation in both the modes. The **read** operation for the file object and the **lookup** operations for the directory and indexed file objects are read-only operations. They are implemented in a way that does not require accessing a remote copy; hence, the concurrent implementation does not affect performance. The first time one of the other operations is executed, *ISIS* requires that a

Table 7.2. Performance of the concurrent implementation

OBJECT TYPE	OPERATION	INVOCATIONS / SEC.					
		1 site		2 sites		6 sites	
		sync.	conc.	sync.	conc.	sync.	conc.
file	**read**	13.0	n.a.	11.0	n.a.	11.0	n.a.
	write	4.2	12.5	1.3	11.0	0.8	9.0
directory	**bind**	2.9	6.3	0.9	6.3	0.6	5.0
	lookup	11.0	n.a.	10.0	n.a.	11.0	n.a.
stack	**push**	2.7	9.2	1.1	9.3	0.5	9.6
	pop	3.3	9.2	1.7	10.5	1.0	8.9
indexed_file	**add**	1.6	3.8	0.5	5.0	0.3	2.3
	lookup	9.5	n.a.	9.5	n.a.	9.5	n.a.

write lock be acquired at remote components. The time taken to acquire the lock is averaged over the remaining 24 invocations, which occur within the same transaction and do not have to wait.

7.5. Discussion

The figures show that the concurrent implementation results in a significant improvement in performance when compared with a synchronous implementation. Performance gains varied from 200% to 1000% in terms of the number of invocations executed per second. Our figures for the concurrent mode are better even in the single site case because the concurrent implementation was used to maintain message routing tables in the type managers. An important observation is that the performance of the synchronous implementation degrades rapidly as the number of sites is increased, while that of the concurrent implementation decreases only slightly. This means that the perceived performance of a replicated object accessible from a number of sites can be made comparable to that of a single-site (non-replicated) object. Although these results are not surprising in the light of the way in which the concurrent implementation works, it is indeed encouraging to see that the constraints of a real-life system do not invalidate our expectations.

One interesting observation that resulted from our tests was that as the rate of performing operations was increased, the buffering of notifications became a bottleneck. This is in keeping with our remark in Chapter 4 that message scheduling could have an important influence on the performance of the concurrent implementation.

These experiments by no means constitute an extensive test of the method. Nevertheless, they suffice to show that considerable performance benefits can be obtained from the concurrent implementation of replicated objects. The measurements include the overhead of the *ISIS* system. Furthermore, because the *ISIS* system is still only a prototype and is built on top of UNIX, it is not tuned for high performance. The multi-process structure imposes a substantial scheduling and inter-process communication overhead. Also, the performance of the remote procedure call connections is suboptimal, primarily because the SUN version of UNIX does not support changes to the IPC buffer size. Even better results can be expected in a system that is fine tuned for high performance.

7.6. Acknowledgements

The performance studies presented in this chapter are largely the result of the work of Ken Birman, who implemented a major portion of the *ISIS* system. Thomas Räuchle implemented a large part of the communication sub-system, and Pat Stevenson constructed a performance measurement tool, which helped greatly in carrying out the tests.

CHAPTER 8

Conclusion

8.1. What have we achieved?

The main result of this work has been to demonstrate that data may be replicated in a distributed system without incurring the cost of latency. We first presented a general model for distributed systems and extended the database concepts of conflicting operations and serializability. We then used this model to show that in a replicated object, operations on remote copies of data may be decoupled from local operations, so that the perceived performance is comparable to that of a non-replicated object. In addition, we showed that the implementation can be done in a way that requires no more messages than a single-site object accessed from more than one site. Finally, we validated our claims by testing out the method on an actual system.

Our work shows that if data are being accessed from multiple sites in a distributed system, then it can be replicated at these sites, while attaining high levels of concurrency. The implication is that in such a system, the only costs that must be incurred in order to obtain high availability and fault-tolerance by replication are the costs of storing more than one copy of data and local processing costs.

8.2. Where do we go from here?

There are two general directions in which this work may be continued. The first relates to the choice of serializability as our correctness criterion. The other deals with the restriction to asynchronous systems.

8.2.1. Correctness constraints

It has been observed that in some situations, serializability is too restrictive a criterion for correctness [21, 30]. We have adopted serializability as our correctness constraint for the following reasons. To begin with, our notion of serializability is more general than in database literature – it is really a requirement that there be some form of atomicity in the system. Atomicity is a natural and simple constraint to require in a distributed system. A closer look at our results, however, reveals that our methods of replicating data depend only on the fact that conflicting operations are ordered relative to each other. They do not require that this ordering occurs in a way that leads to serializability. This opens the possibility that the methods can be used without change in a system where a weaker form of correctness is being enforced. Characterizing this weaker form of correctness formally and demonstrating situations where such a correctness constraint may be useful remains an area for future work. The inverse problem – that of adapting the methods given here to suit a system with a given (weaker) correctness constraint – is another such area.

8.2.2. What if the system is synchronous?

In an asynchronous system, the only way in which a site becomes aware of the completion of an event at another site is by receiving a message from the latter site. Our implementations make use of these messages to transfer information needed to keep the copies of replicated data consistent. In a system where a synchronized clock is available, the mere fact that a certain period of time has elapsed could be used to infer that an event has taken place at a remote site. For example, if all sites make backups of their respective file systems at 5 o'clock every day, and if backup operations take no longer than half an hour, then at 5:30 every site knows that every other site must have completed a backup for the day. Thus in a synchronous system, information can be gathered without explicit message transfer. In [33], Lamport discusses the relation between using synchronized clocks and using messages to order events.

An assumption in the example above is that the behavior of every site is "known" to every other site, otherwise they could not have concluded that the backups were done. The concept of "knowledge" in a distributed system has been studied in [13, 24]. It is shown that the information available to a site in a distributed system is a consequence of "common knowledge" at the time the system is initialized and of knowledge gained by the transfer of messages between sites. It would be interesting to study whether the results presented here can be expressed on the basis of "knowledge transfer" rather than message transfer, and thus made applicable to synchronous systems.

List of References

1. ALLCHIN, J. E., AND MCKENDRY, M. S., Synchronization and recovery for actions. *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983, 31-44.

2. ANDREWS, G. R., AND SCHNEIDER, F. B., Concepts and notations for concurrent programming. *ACM Computing Surveys 15*, 1 (March 1983), 3-44.

3. BERNSTEIN, P. A., AND GOODMAN, N., Concurrency control in distributed database systems. *ACM Computing Surveys 13*, 2 (June 1981), 185-221.

4. BERNSTEIN, P. A., AND GOODMAN, N., Concurrency control and recovery for replicated distributed databases. Technical Report TR-20-83, Aiken Computing Laboratory, Harvard University, 1983.

5. BERNSTEIN, P. A., SHIPMAN, D. W., AND WONG, W. S., Formal aspects of serializability. *IEEE Transactions on Software Engineering*, SE-5 (May 1979), 203-215.

6. BIRMAN, K., Replication and fault-tolerance in the *ISIS* system. *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Orcas Island, WA, Dec. 1985.

7. BIRMAN, K., DIETRICH, W., EL-ABBADI, A., JOSEPH, T. A., RAEUCHLE, T., An overview of the *ISIS* project. Technical Report, TR 84-642, Department of Computer Science, Cornell University, Oct. 1984.

8. Birman, K., Joseph, T. A., Reliable communication in an unreliable environment. Technical Report, TR 85-694, Department of Computer Science, Cornell University, Aug. 1985.

9. Birman, K., Joseph, T. A., Raeuchle, T., Extending resilient object types efficiently. *2nd GI NTG GMR Conference on Fault-Tolerant Computing Systems*, Bonn, West Germany, Sept. 1984.

10. Birman, K., Joseph, T. A., Raeuchle, T., El-Abbadi, A., Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering SE-11*, 6 (June 1985), 502-508.

11. Cabrera, L. F., Hunter, E., Karels, M., and Mosher, D. A user-process oriented performance study of Ethernet networking under Berkeley UNIX 4.2BSD. Report No. UCB/CSD 84/217, Computer Science Division (EECS), University of California, Berkeley, California, Dec. 1984.

12. Casanova, M. A., The concurrency control problem for database systems. *Lecture Notes in Computer Science 116*, Springer-Verlag, 1981.

13. Chandy, M., and Misra, J., How processes learn. *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Canada, Aug. 1985, 204-214.

14. Chang, J., and Maxemchuk, N. F., Reliable Broadcast Protocols. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 251-273.

15. CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D., Atomic broadcast: from simple message diffusion to Byzantine agreement. IBM Research Report, RJ 4540 (48688), Oct. 1984.

16. DIJKSTRA, E. W., Cooperating Sequential Processes. *Programming Languages*, F. Genuys (Ed.), Academic Press, New York, 1968.

17. EAGER, D. L., AND SEVCIK, K. C., Achieving robustness in distributed database systems. *ACM Transactions on Database Systems 8*, 3 (Sept. 1983), 354-381.

18. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L., The notion of consistency and predicate locking in a relational database system. *Communications of the ACM 19*, 11 (Nov. 1976), 624-633.

19. GARCIA-MOLINA, H., Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems 8*, 2 (June 1983), 186-213.

20. GRAY, J. N., The transaction concept: virtues and limitations. *Proceedings of the 7th International Conference on Very Large Data Bases*, Sept. 1981, 144-154.

21. GRAY, J. N., LORIE, R. A., AND PUTZOLU, G. R., Granularity of locks in a shared data base. *Proceedings of the International Conference on Very Large Data Bases 1*, 1 (1975), 428-451.

22. GUTTAG, J. V., AND HORNING, J. J., The algebraic specification of abstract data types. *Acta Informatica 10*, 1978, 27-52.

23. Hadzilacos, V., Issues of fault tolerance in concurrent computations. Ph. D. thesis, TR-11-84, Division of Applied Sciences, Harvard University, June 1984.

24. Halpern, J. Y., and Moses, Y. O., Knowledge and common knowledge in a distributed environment. *Proceedings of the 3rd Annual ACM Conference on Principles of Distributed Computing*, Vancouver, Canada, Aug. 1984, 50-61.

25. Herlihy, M. P., Replication methods for abstract data types. Ph. D. thesis, MIT/LCS/TR-319, Laboratory of Computer Science, Massachusetts Institute of Technology, May 1984.

26. Hoare, C. A. R., Monitors: an operating system structuring concept. *Communications of the ACM 17*, 10 (Oct. 1974), 549-557.

27. Hoare, C. A. R., Specifications, programs and implementations. Technical Monograph, PRG-29, Oxford University Computing Laboratory, June 1982.

28. Joseph, T. A., and Birman, K., Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems 4*, 1 (Feb. 1986).

29. Kung, H. T., and Papadimitriou, C. H., An optimality theory of concurrency control for databases. *Acta Informatica 19*, 1 (1983), 1-11.

30. Lamport, L., Towards a theory of correctness of multi-user data base systems. Massachusetts Computer Associates Inc. Report, CA-7610-0712, Oct. 1976.

31. Lamport, L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558-565.

32. Lamport, L., Specifying concurrent program modules. Computer Science Laboratory, SRI International, June 1981.

33. Lamport, L., Using time instead of timeout in fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems 6*, 2 (April 1984), 254-280.

34. Lazowska, E. D., Levy, H. M., Almes, G. T., Fischer, M. J., Fowler, R. J., and Vestal, S. C., The architecture of the Eden system. *Proceedings of the 8th Symposium on Operating Systems Principles*, Dec. 1981, 148-159.

35. Liskov, B., and Scheifler, R., Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

36. Metcalfe, R. M., and Boggs, D. R., Ethernet: distributed packet switching for local computer networks. *Communications of the ACM 19*, 7 (July 1976), 395-404.

37. Papadimitriou, C. H., Serializability of concurrent database updates. *Journal of the ACM 26*, 4 (Oct. 1979), 631-653.

38. Papadimitriou, C. H., Bernstein, P. A., and Rothnie, J. B., Computational problems related to database concurrency control. *Proceedings of Conference on Theoretical Computer Science*, Waterloo, Canada, 1977, 275-282.

39. Schlichting, R., and Schneider, F. B., Fail-stop processors: an approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems 1*, 3 (August 1983), 222-238.

40. SCHNEIDER, F. B., Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems 2*, 3 (May 1984), 145-154.

41. SCHNEIDER, F. B., GRIES, D., AND SCHLICHTING, R., Fault-tolerant broadcast. *Science of Computer Programming 4*, 1984, 1-15.

42. SCHWARZ, P. M., AND SPECTOR, A. Z., Synchronizing shared abstract data types. *ACM Transactions on Computer Systems 2*, 3 (August 1984), 223-250.

43. STEARNS, R. E., LEWIS, P. M., AND ROSENKRANTZ, D. J., Concurrency controls for database systems. *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, 1976, 19-32.

44. STONEBRAKER, M., Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering SE-5*, 3 (May 1979), 188-194.

45. TRAIGER, I. L., GRAY, J. N., GALTIERI, C. A., AND LINDSAY, B. G., Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems 7*, 3 (Sept. 1982), 323-342.

46. WEIHL, W. E., Specification and implementation of atomic data types. Ph. D thesis, MIT/LCS/TR-314, Laboratory of Computer Science, Massachusett Institute of Technology, March 1984.

47. WUU, G. T. J., AND BERNSTEIN, A. J., Efficient solutions to the replicated log and dictionary problems. *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, Aug. 1984.

48. YANNAKAKIS, M., Issues of correctness in database concurrency control by locking. *Proceedings of the 13th ACM SIGACT Symposium on Theory of Computing*, Milwaukee, 1981, 363-367.

Index of Definitions